International Journal of Reconfigurable Computing

# High-Performance Reconfigurable Computing

Guest Editors: Khaled Benkrid, Esam El-Araby, Miaoqing Huang, Kentaro Sano, and Thomas Steinke

# High-Performance Reconfigurable Computing

International Journal of Reconfigurable Computing

# High-Performance Reconfigurable Computing

Guest Editors: Khaled Benkrid, Esam El-Araby,
Miaoqing Huang, Kentaro Sano, and Thomas Steinke

# Editorial Board

# Contents

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 104963, 2 pages
doi:10.1155/2012/104963

# *Editorial*
# High-Performance Reconfigurable Computing

**Khaled Benkrid,[1] Esam El-Araby,[2] Miaoqing Huang,[3] Kentaro Sano,[4] and Thomas Steinke[5]**

[1] *School of Engineering, The University of Edinburgh, Edinburgh EH9 3JL, UK*
[2] *Electrical Engineering and Computer Science, The Catholic University of America, Washington, DC 20064, USA*
[3] *Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR 72701, USA*
[4] *Graduate School of Information Sciences, Tohoku University, 6-6-01 Aramaki Aza Aoba, Sendai 980-8579, Japan*
[5] *Zuse-Institut Berlin (ZIB), Takustraße 7, 14195 Berlin-Dahlem, Germany*

Correspondence should be addressed to Khaled Benkrid, k.benkrid@ed.ac.uk

Received 28 February 2012; Accepted 28 February 2012



This special issue presents some of the latest developments in the burgeoning area of high-performance reconfigurable computing (HPRC) which aims to harness the high-performance and low power of reconfigurable hardware in the forms of field programmable gate arrays (FPGAs) in high-performance computing (HPC) applications.

The issue starts with three widely popular HPC applications, namely, financial computing, bioinformatics and computational biology, and high-throughput data search. First, Starke et al. from the University of Kiel in Germany present the use of a massively parallel FPGA platform, called RIVYERA, in the high-performance and low-power optimization of investment strategies. The authors demonstrate an FPGA-based implementation of an investment strategy algorithm that considerably outperforms a single CPU node in terms of raw processing power and energy efficiency. Furthermore, it is shown that the implemented optimized investment strategy outperforms a buy-and-hold strategy. Then, Vanderbauwhede et al. from Glasgow University and the University of Massachusetts Lowell propose a design for the scoring part of a high-throughput real-time search application on FPGAs. The authors use a low-latency Bloom filter to realize high-performance information filtering. An analytical model of the application throughput is built around the Bloom filter. The experimental results on the Novo-G reconfigurable supercomputer demonstrate a 20× speedup compared with a software implementation on a 3.4 GHz Intel Core i7 processor. After that, Eusse et al. from the University of Brasilia and the Federal University of Mato Grosso do Sul in Brazil present an FPGA-accelerated protein sequence analysis solution. The authors integrate the concept of divergence to the Viterbi algorithm used in the HMMER program suite for biological sequence alignment, in order to reduce the area of the score matrix in which the trace-back alignment is made. This technique leads to large speedups (182×) compared to nonaccelerated pure software processing.

The issue then presents a number of architectural concerns in the design of HPRC systems, namely: reconfigurable hardware architecture, communication network design, and arithmetic design. First, Wan et al. from the University of Illinois at Urbana Champaign and Magma Design Automation Inc. in the USA present a coarse-grained reconfigurable architecture (CGRA) with a Fast Data Relay (FDR) mechanism to enhance its performance. This is achieved through multicycle data transmission concurrent with computation, and effective reduction of communication traffic congestion. The authors also propose compiler techniques to efficiently utilize the FDR feature. The experimental results for various multimedia applications show that FDR combined with the new compiler delivers up to 29% and 21% higher performance than other CGRAs: ADRES and RCP, respectively. The following paper by Schmidt et al. from the University of Southern California and the University of North Carolina present an integrated on-chip/off-chip network with MPI-style point-to-point message, implemented on an all-FPGA computing cluster. The most salient differences between the network architecture presented in this paper and state-of-the-art Network-on-Chip (NoC) architectures is the use of a single full-crossbar switch. The results are different from

other efforts due to several reasons. First, the implementation target is the programmable logic of an FPGA. Second, most NoCs assume that a full crossbar is too expensive in terms of resources while within the context of an HPRC system the programmable logic resources are fungible. The authors justify their focus on the network performance by the fact that overall performance is limited by the bandwidth off the chip rather than by the mere number of compute cores on the chip. After that, El-Araby et al. from the Catholic University of America, Universidad Autonoma de Madrid, and the George Washington University present a technique for the acceleration of arbitrary-precision arithmetic on HPRC systems. Efficient support of arbitrary-precision arithmetic in very large science and engineering simulations is particularly important as numerical nonrobustness becomes increasingly an issue in such applications. Today's solutions for arbitrary-precision arithmetic are usually implemented in software and performance is significantly reduced as a result. In order to reduce this performance gap, the paper investigates the acceleration of arbitrary-precision arithmetic on HPRC systems.

The special issue ends with two papers which present reconfigurable hardware in HPC in the context of other computer technologies. First, Benkrid et al. from the University of Edinburgh, Scotland, and the University of Arizona, USA, present a comparative study of FPGAs, Graphics Processing Units (GPUs), IBM's Cell BE, and General Purpose Processors (GPPs) in the design and implementation of a biological sequence alignment application. Using speed, energy consumption, in addition to purchase and development costs, as comparison criteria, the authors argue that FPGAs are high-performance economic solutions for sequence alignment applications. In general, however, they argue that FPGAs need to achieve at least two orders of magnitude speedup compared to GPPs and one order of magnitude speedup compared to GPUs to justify their relatively longer development times and higher purchase costs. The following paper by Inta et al. from the Australian National University presents an off-the-shelf CPU, GPU, and FPGA heterogeneous computing platform, called Chimera, as a potential high-performance economic solution for certain HPC applications. Motivated by computational demands in the area of astronomy, the authors propose the Chimera platform as a viable alternative for many common computationally bound problems. Advantages and challenges of migrating applications to such heterogeneous platforms are discussed by using demonstrator applications such as Monte Carlo integration and normalized cross-correlation. The authors show that the most significant bottleneck in multidevice computational pipelines is the communications interconnect.

We hope that this special issue will serve as an introduction to those who have newly joined, or are interested in joining, the HPRC research community as well as provide specialists with a sample of the latest developments in this exciting research area.

*Khaled Benkrid*
*Esam El-Araby*
*Miaoqing Huang*
*Kentaro Sano*
*Thomas Steinke*

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 925864, 14 pages
doi:10.1155/2012/925864

*Research Article*

# A Convolve-And-MErge Approach for Exact Computations on High-Performance Reconfigurable Computers

**Esam El-Araby,[1] Ivan Gonzalez,[2] Sergio Lopez-Buedo,[2] and Tarek El-Ghazawi[3]**

[1] *Department of Electrical Engineering and Computer Science, The Catholic University of America, Washington, DC 20064, USA*
[2] *Departments of Computer Engineering at Escuela Politecnica Superior of Universidad Autonoma de Madrid, 28049 Madrid, Spain*
[3] *Department of Electrical and Computer Engineering, The George Washington University, Washington, DC 20052, USA*

Correspondence should be addressed to Esam El-Araby, aly@cua.edu

Received 31 October 2011; Revised 1 February 2012; Accepted 13 February 2012

Academic Editor: Thomas Steinke



This work presents an approach for accelerating arbitrary-precision arithmetic on high-performance reconfigurable computers (HPRCs). Although faster and smaller, fixed-precision arithmetic has inherent rounding and overflow problems that can cause errors in scientific or engineering applications. This recurring phenomenon is usually referred to as numerical nonrobustness. Therefore, there is an increasing interest in the paradigm of exact computation, based on arbitrary-precision arithmetic. There are a number of libraries and/or languages supporting this paradigm, for example, the GNU multiprecision (GMP) library. However, the performance of computations is significantly reduced in comparison to that of fixed-precision arithmetic. In order to reduce this performance gap, this paper investigates the acceleration of arbitrary-precision arithmetic on HPRCs. A Convolve-And-MErge approach is proposed, that implements virtual convolution schedules derived from the formal representation of the arbitrary-precision multiplication problem. Additionally, dynamic (nonlinear) pipeline techniques are also exploited in order to achieve speedups ranging from 5x (addition) to 9x (multiplication), while keeping resource usage of the reconfigurable device low, ranging from 11% to 19%.

## 1. Introduction

Present-day computers built around fixed-precision components perform integer and/or floating point arithmetic operations using fixed-width operands, typically 32 and/or 64 bits wide. However, some applications require larger precision arithmetic. For example, operands in public-key cryptography algorithms are typically thousands of bits long. Arbitrary-precision arithmetic is also important for scientific and engineering computations where the roundoff errors arising from fixed-precision arithmetic cause convergence and stability problems. Although many applications can tolerate fixed-precision problems, there is a significant number of other applications, such as finance and banking, in which numerical overflow is intolerable. This recurring phenomenon is usually referred to as numerical nonrobustness [1]. In response to this problem, exact computation, based on exact/arbitrary-precision arithmetic, was first introduced in 1995 by Yap and Dube [2] as an emerging numerical computation paradigm. In arbitrary-precision arithmetic, also known as *bignum* arithmetic, the size of operands is only limited by the available memory of the host system [3, 4].

Among other fields, arbitrary-precision arithmetic is used, for example, in computational metrology and coordinate measuring machines (CMMs), computation of fundamental mathematical constants such as $\pi$ to millions of digits, rendering fractal images, computational geometry, geometric editing and modeling, and constraint logic programming (CLP) languages [1–5].

In the earlier days of computers, there were some machines that supported arbitrary-precision arithmetic in hardware. Two examples of these machines were the IBM 1401 [6] and the Honeywell 200 Liberator [7] series. Nowadays, arbitrary-precision arithmetic is mostly implemented in software, perhaps embedded into a computer compiler. Over the last decade, a number of *bignum* software packages have been developed. These include the GNU multiprecision (GMP) library, CLN, LEDA, Java.math, BigFloat, BigDigits,

and Crypto++ [4, 5]. In addition, there exist stand-alone application software/languages such as PARI/GP, Mathematica, Maple, Macsyma, dc programming language, and REXX programming language [4].

Arbitrary-precision numbers are often stored as large-variable-length arrays of digits in some base related to the system word-length. Because of this, arithmetic performance is slower compared to fixed-precision arithmetic which is closely related to the size of the processor internal registers [2]. There have been some attempts for hardware implementations. However, those attempts usually amounted to specialized hardware for small-size discrete multiprecision and/or to large-size fixed-precision [8–12] integer arithmetic rather than to real arbitrary-precision arithmetic.

High-performance reconfigurable computers (HPRCs) have shown remarkable results in comparison to conventional processors in those problems requiring custom designs because of the mismatch with operand widths and/or operations of conventional ALUs. For example, speedups of up to 28,514 have been reported for cryptography applications [13, 14], up to 8,723 for bioinformatics sequence matching [13], and up to 32 for remote sensing and image processing [15, 16]. Therefore, arbitrary-precision arithmetic seemed to be a good candidate for acceleration on reconfigurable computers.

This work explores the use of HPRCs for arbitrary-precision arithmetic. We propose a hardware architecture that is able to implement addition, subtraction and multiplication, as well as convolution, on arbitrary-length operands up to 128 ExibiByte. The architecture is based on virtual convolution scheduling. It has been validated on a classic HPRC machine, the SRC-6 [17] from SRC Computers, showing speedups ranging from 2 to 9 in comparison to the portable version of the GMP library. This speedup is in part attained due to the dynamic (nonlinear) pipelining techniques that are used to eliminate the effects of deeply pipelined reduction operators.

The paper is organized as follows. Section 2 presents a short overview of HPRC machines. The problem is formulated in Section 3. Section 4 describes the proposed approach and architecture augmented with a numerical example for illustrating the details of the proposed approach. Section 5 shows the experimental work. Implementation details are also given in Section 5, as well as performance comparison to the SW version of GMP. Finally, Section 6 presents the conclusions and future directions.

## 2. High Performance Reconfigurable Computing

In the recent years, the concept of high-performance reconfigurable computing has emerged as a promising alternative to conventional processing in order to enhance the performance of computers. The idea is to accelerate a parallel computer with reconfigurable devices such as FPGAs where a custom hardware implementation of the critical sections of the code is performed in the reconfigurable device. Although the clock frequency of the FPGA is typically one order of magnitude less than the one of high-end microprocessors, significant speedups are obtained due to the increased parallelism of hardware. This performance is especially important for those algorithms not matching the architecture of conventional microprocessors, because of either the operand lengths (e.g., bioinformatics) or the operations performed (e.g., cryptography). Moreover, the power consumption is reduced in comparison to conventional platforms, and the use of reconfigurable devices brings flexibility closer to that of SW, as opposed to other HW-accelerated solutions such as ASICs. In other words, the goal of HPRC machines is to achieve the synergy between the low-level parallelism of hardware with the system-level parallelism of high-performance computing (HPC) machines.

In general, HPRCs can be classified as either nonuniform node uniform systems (NNUSs) or uniform node nonuniform systems (UNNSs) [13]. NNUSs consist of only one type of nodes. Nodes are heterogeneous containing both FPGAs and microprocessors. FPGAs are connected directly to the microprocessors inside the node. On the other hand, UNNS nodes are homogeneous containing either FPGAs or microprocessors which are linked via an interconnection network. The platform used in this paper, SRC-6 [17], belongs to the second category.

SRC-6 platform consists of one or more general-purpose microprocessor subsystems, one or more MAP reconfigurable processor subsystems, and global common memory (GCM) nodes of shared memory space [17]. These subsystems are interconnected through a Hi-Bar Switch communication layer; see Figure 1. Multiple tiers of the Hi-Bar Switch can be used to create large-node count scalable systems. Each microprocessor board is based on 2.8 GHz Intel Xeon microprocessors. Microprocessors boards are connected to the MAP boards through the SNAP interconnect. The SNAP card plugs into the memory DIMM slot on the microprocessor motherboard to provide higher data transfer rates between the boards than the less efficient but common PCI solution. The peak transfer rate between a microprocessor board and the MAP board is 1600 MB/sec. Hardware architecture of the SRC-6 MAP processor is shown in Figure 1. The MAP Series C board is composed of one control FPGA and two user FPGAs, all Xilinx Virtex II-6000-4. Additionally, each MAP unit contains six interleaved banks of on-board memory (OBM) with a total capacity of 24 MB. The maximum aggregate data transfer rate among all FPGAs and on-board memory is 4800 MB/s. The user FPGAs are configured in such a way that one is in the master mode and the other is in the slave mode. The two FPGAs of a MAP are directly connected using a bridge port. Furthermore, MAP processors can be chained together using a chain port to create an array of FPGAs.

## 3. Problem Formulation

Exact arithmetic uses the four basic arithmetic operations $(+, -, \times, \div)$ over the rational field $\mathbf{Q}$ to support exact computations [1–5]. Therefore, the problem of exact computation is reduced to implementing these four basic arithmetic operations with arbitrary-sized operands.

Table 1: Computational complexity of arithmetic operations [18].

| Operation | Input | Output | Algorithm | Complexity |
|---|---|---|---|---|
| Addition | Two $n$-digit numbers | One $(n+1)$-digit number | Basecase/Schoolbook | $O(n)$ |
| Subtraction | Two $n$-digit numbers | One $(n+1)$-digit number | Basecase/Schoolbook | $O(n)$ |
| Multiplication | Two $n$-digit numbers | One $2n$-digit number | Basecase/Schoolbook | $O(n^2)$ |
| | | | Karatsuba | $O(n^{1.585})$ |
| | | | 3-way Toom-Cook | $O(n^{1.465})$ |
| | | | $k$-way Toom-Cook | $O(n^{1+\varepsilon})$, $\varepsilon > 0$ |
| | | | Mixed-level Toom-Cook | $O(n(\log n)2^{\sqrt{(2\log n)}})$ |
| | | | Schösnhage-Strassen | $O(n(\log n)(\log\log n))$ |
| Note: The complexity of multiplication will be referred to as $M(n)$ in the following | | | | |
| Division | Two $n$-digit numbers | One $n$-digit number | Basecase/Schoolbook | $O(n^2)$ |
| | | | Newton's method | $O(M(n))$ |
| | | | Goldschmidt | $O(M(n))$ |
| Square root | One $n$-digit number | One $n$-digit number | Newton's method | $O(M(n))$ |
| | | | Goldschmidt | $O(M(n))$ |
| Polynomial evaluation | $n$ fixed-size polynomial coefficients | One fixed size | Horner's method | $O(n)$ |
| | | | Direct evaluation | $O(n)$ |

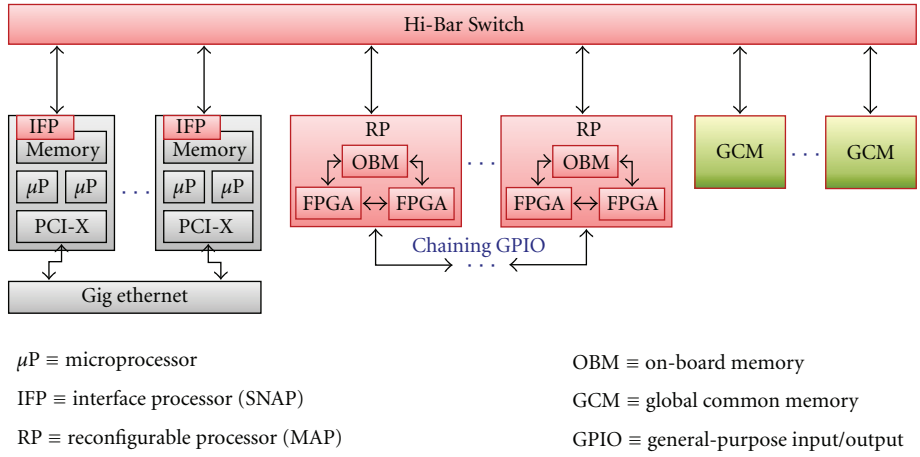



Figure 1: Hardware architecture of SRC-6.

The asymptotic computational complexity of each operation depends on the bit length of operands [5, 18], see Table 1. Our formal representation of the problem will consider only the multiplication operation. This is based on the fact that multiplication is a core operation from which the other basic arithmetic operations can be easily derived, for example, division as a multiplication using Newton-Raphson's approximation or Goldschmidt algorithm [19, 20]. Our proposed arithmetic unit can perform arbitrary-precision addition, subtraction, multiplication, as well as convolution operations. We decided to follow the Basecase/Schoolbook algorithm. Although this algorithm is not the fastest algorithm having a complexity of $O(n^2)$, see Table 1, it is the simplest and most straightforward algorithm with the least overhead. In addition, this algorithm is usually the starting point for almost all available software implementations of arbitrary-precision arithmetic. For example, in the case of GMP, the Basecase algorithm is used up to a predetermined operand size threshold, 3000–10,000 bit length depending on the underlying microprocessor architecture, beyond which the software adaptively switches to a faster algorithm [20, 21].

The main challenge of implementing arbitrary-precision arithmetic in HPRC machines is the physical/spatial limitations of the reconfigurable device. In other words, the reconfigurable device (FPGA) has limited physical resources, which makes it unrealistic to accommodate for the resource requirements of arbitrarily large-precision arithmetic operators. Therefore, the problem can be formulated as given a fixed-precision arithmetic unit, for example, $p$-digit by $p$-digit multiplier, how to implement an arbitrary-precision arithmetic unit, for example, arbitrary large-variable-size $m_1$-digit by $m_2$-digit multiplier. Typically, $p$ is dependent on the underlying hardware word-length, for example, 32-bit

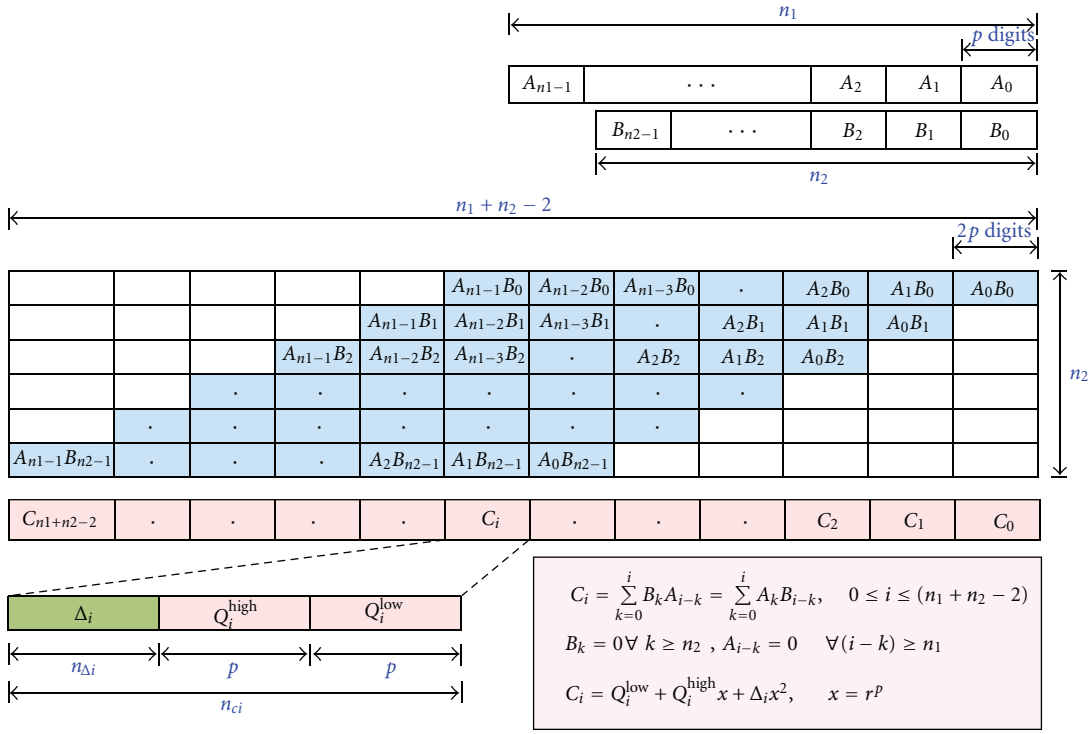



Figure 2: Multiplication matrix of high-precision numbers.

or 64-bit. In achieving this objective, our approach is based on leveraging previous work and concepts that were introduced for solving similar problems. For example, Tredennick and Welch [22] proposed architectural solutions for variable-length byte string processing. Similarly, Olariu et al. [23] formally analyzed and proposed solutions for the problem of sorting arbitrary large number of items using a sorting network of small fixed I/O size. Finally, ElGindy and Ferizis [24] investigated the problem of mapping recursive algorithms on reconfigurable hardware.

## 4. Approach and Architectures

*4.1. Formal Problem Representation.* An arbitrary-precision $m$-digit number in arbitrary numeric base $r$ can be represented by

$$A = a_0 + a_1 r + a_2 r^2 + \cdots + a_{m-1} r^{m-1},$$
$$A = \sum_{j=0}^{m-1} a_j r^j, \quad 0 \le a_j < r. \tag{1}$$

It can also be interpreted as an $n$-digit number with base $r^p$, where $p$ is dependent on the underlying hardware word-length, for example, 32-bit or 64-bit. This is represented by (2) as follows:

$$A = \sum_{i=0}^{n-1} \sum_{j=ip}^{(i+1)p-1} a_j r^j = \sum_{i=0}^{n-1} \sum_{k=0}^{p-1} a_{k+ip} r^{k+ip}$$
$$= \sum_{i=0}^{n-1} \left[ \sum_{k=0}^{p-1} a_{k+ip} r^k \right] r^{ip} = \sum_{i=0}^{n-1} A_i r^{ip},$$
$$\text{where } A_i = \sum_{k=0}^{p-1} a_{k+ip} r^k, \quad n = \left\lceil \frac{m}{p} \right\rceil. \tag{2}$$

Multiplication, accordingly, can be formulated as shown in Figure 2 and expressed by (3). In other words, as implied by (4a), multiplication of high-precision numbers can be performed through two separate processes in sequence. The first is a low fixed precision, that is, $p$-digits, multiply-accumulate (MAC) process for calculating the coefficients/partial products $C_i$s as given by (4b). This is followed by a merging process of these coefficients/partial products into a final single high-precision product as given by (4a). Equation (4b) shows that the coefficients $C_i$s can be represented at minimum by $2p$-digit precision.

The extra digits are due to the accumulation process. Therefore, $C_i$s can be expressed as shown by (4c);

$$A = \sum_{i=0}^{n_1-1} A_i r^{ip} = \sum_{i=0}^{n_1-1} A_i x^i = A(x),$$
$$B = \sum_{i=0}^{n_2-1} B_i r^{ip} = \sum_{i=0}^{n_2-1} B_i x^i = B(x),$$
$$C = AB = A(x) \cdot B(x) = C(x), \tag{3}$$
$$\text{where } n_1 = \left\lceil \frac{m_1}{p} \right\rceil, \quad n_2 = \left\lceil \frac{m_2}{p} \right\rceil, \quad x = r^p.$$

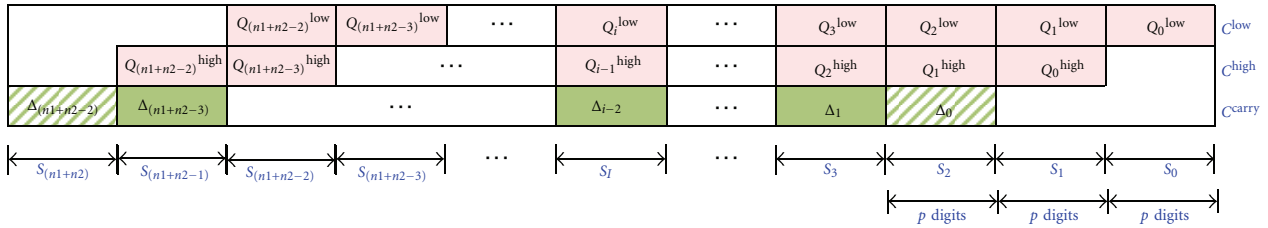



Figure 3: Merging schedule.

$$C(x) = C_0 + C_1x + C_2x^2 + C_3x^3 + \cdots + C_{(n_1+n_2-2)}x^{(n_1+n_2-2)}, \tag{4a}$$

where

$$C_i = \sum_{k=0}^{i} B_kA_{i-k} = \sum_{k=0}^{i} A_kB_{i-k}, \quad 0 \le i \le (n_1+n_2-2),$$
$$B_k = 0, \quad \forall k \ge n_2,$$
$$A_{i-k} = 0, \quad \forall (i-k) \ge n_1, \tag{4b}$$

$$C_i = Q_i^{low} + Q_i^{high}x + \Delta_i x^2, \quad x = r^p. \tag{4c}$$

*4.2. Multiplication as a Convolve-And-MErge (CAME) Process.* It can be easily noticed that the coefficients $C_i$s given by (4b) are in the form of a convolution sum. This led us to believe that virtualizing the convolution operation and using it as a scheduling mechanism will be a straightforward path for implementing multiplication and hence the remaining arithmetic operations. The different sub operands, that is, $A$s and $B$s, being stored in the system memory, will be accessed according to the convolution schedule and passed to the MAC process. The outcome of the MAC process is then delivered to the merging process which merges the partial products, according to another merging schedule, into the final results. The final results are then scheduled back into the system memory according to the same convolution schedule.

The convolution schedule, on one hand, can be derived from (4b). It is simply a process that generates the addresses/indexes for $A$s, $B$s, and $C$s governed by the rules given in (4b). On the hand, the merging schedule can be derived from (5) which results from substituting (4c) into (4a). Figure 3 shows the merging schedule as a high-precision addition of three components. The first component is simply a concatenation of all the first $p$-digits of the MAC output. The second component is a $p$-digit shifted concatenation of all the second $p$-digits of the MAC output. Finally, the third component is a $2p$-digit shifted concatenation of all the third $p$-digits of the MAC output:

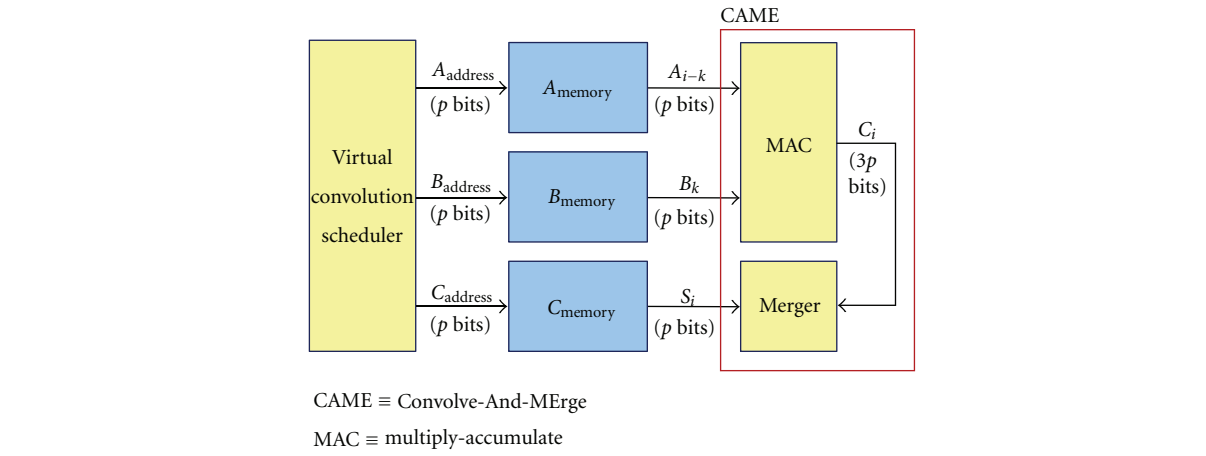
$$C(x) = \sum_{i=0}^{n_1+n_2-2} Q_i^{low}x^i + \left(\sum_{i=0}^{n_1+n_2-2} Q_i^{high}x^i\right) \cdot x + \left(\sum_{i=0}^{n_1+n_2-2} \Delta_i x^i\right) \cdot x^2,$$
$$C(x) \equiv C^{low} + C^{high} \cdot x + C^{carry} \cdot x^2,$$
$$\text{where } C^{low} = \sum_{i=0}^{n_1+n_2-2} Q_i^{low}x^i, \quad C^{high} = \sum_{i=0}^{n_1+n_2-2} Q_i^{high}x^i, \quad C^{carry} = \sum_{i=0}^{n_1+n_2-2} \Delta_i x^i. \tag{5}$$

The merging schedule, as described above, is a high-precision schedule which will work only if the merging process is performed after the MAC process has finished completely. Given the algorithm complexity $O(n^2)$ and allowing the two processes to work sequentially one after another would dramatically impact the performance. However, modifying the merging process to follow a small-fixed-precision scheduling scheme that works in parallel and in synchrony with the MAC process would bring back the performance to its theoretical complexity $O(n^2)$. The modified merging scheme can be very easily derived either from (5) or Figure 3 resulting in (6):

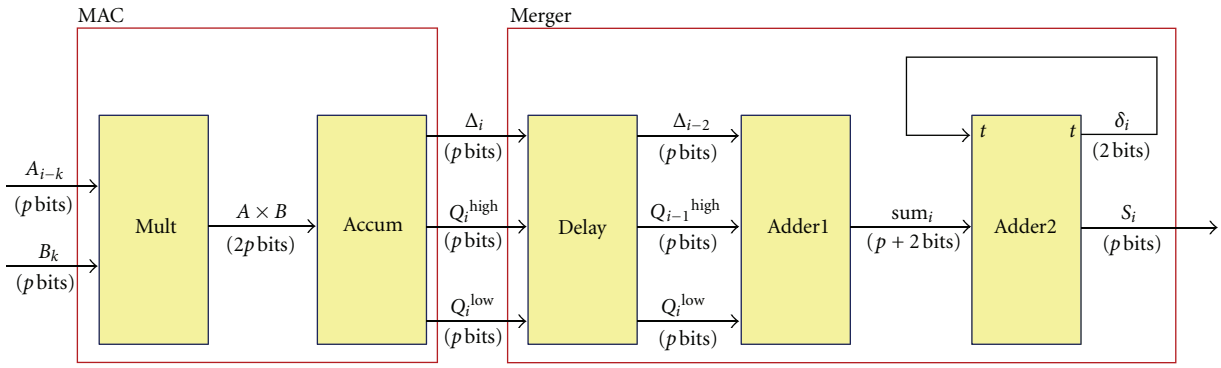
$$S_i = \delta_{i-1} + Q_i^{low} + Q_{i-1}^{high} + \Delta_{i-2}, \quad i = 0, 1, 2, \ldots, (n_1+n_2),$$
$$\delta_i = S_i \cdot x^{-1} = S_i \cdot r^{-p} = \mathrm{SHR}(S_i, p \text{ digits}), \tag{6}$$
$$Q_k^{low} = Q_k^{high} = \Delta_k = 0 \quad \forall k < 0,\ k > (n_1+n_2-2),$$
$$\delta_k = 0 \quad \forall k < 0,\ k \ge (n_1+n_2).$$

This would mean that, as soon as the MAC process finishes one partial result, $C_i$ in $3p$-digit precision, the merging process, in-place, produces a final partial result $S_i$ in $p$-digit precision, see Figure 4. This precision matches the word-length of the supporting memory system which allows easy storage of the final result without stalling either the MAC or the merging process. The merging process registers the remaining high-precision digits for use in subsequent calculations of $S_i$s.

In addition to performing multiplication, the derived architecture in Figure 4 can also natively perform the convolution operation for sequences of arbitrary size. This is because the MAC process generates the coefficients in (4b) according to a convolution schedule and in fact they are the direct convolution result. In other words, only the MAC process is needed for the convolution operation. Furthermore, the same unit can be used to perform addition and/or




(a) Architecture hierarchy and memory requirements




(b) Arithmetic unit architecture

Figure 4: CAME architecture.

subtraction by passing the input operands directly to the merging process without going through the MAC process.

*4.3. Illustrative Example.* In order to show the steps of the proposed methodology, we will consider multiplying two decimal numbers, $A = 987654321$ and $B = 98765$. We will assume that the underlying hardware word-length is 2 digits. The formal representation for this example is

$$r = 10, \qquad p = 2 \Longrightarrow x = r^p = 100,$$

$$A = 987654321 \Longrightarrow m_1 = 9, \qquad n_1 = \left\lceil \frac{m_1}{p} \right\rceil = \left\lceil \frac{9}{2} \right\rceil = 5,$$

$$B = 98765 \Longrightarrow m_2 = 5, \qquad n_2 = \left\lceil \frac{m_2}{p} \right\rceil = \left\lceil \frac{5}{2} \right\rceil = 3,$$

$$A(x) = \sum_{i=0}^{5-1} A_i x^i = A_4 x^4 + A_3 x^3 + A_2 x^2 + A_1 x + A_0,$$

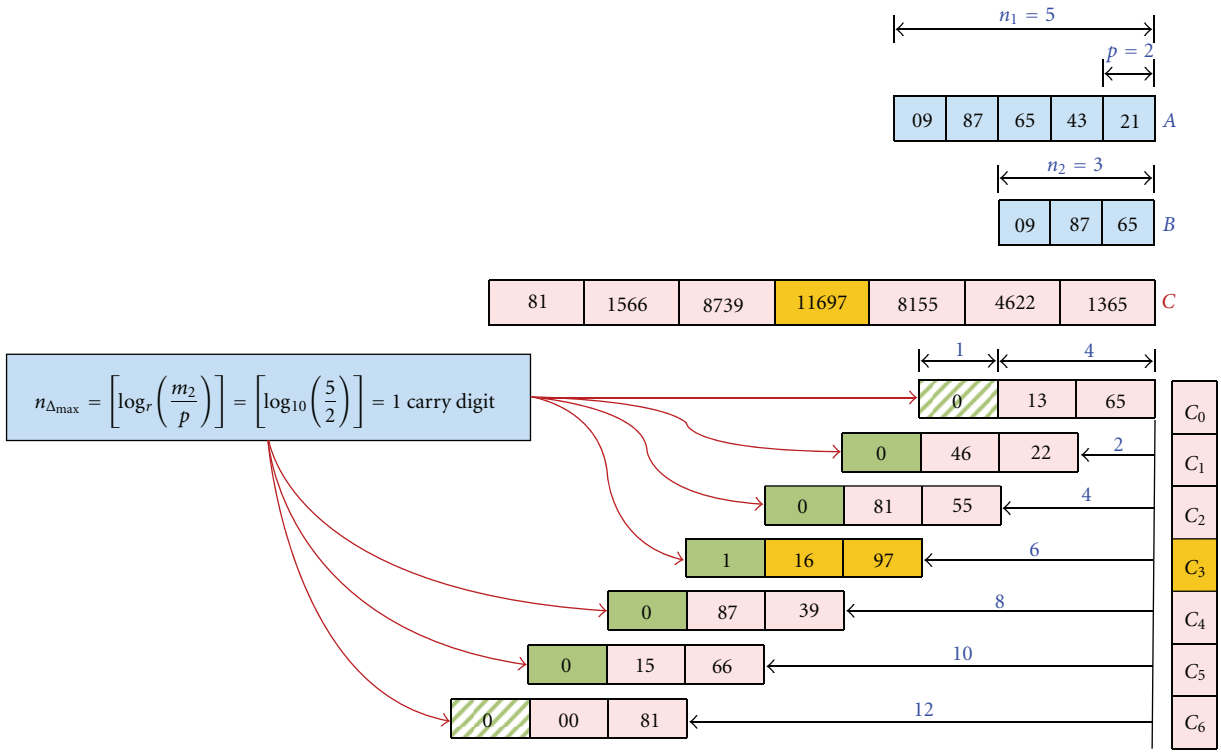
$$A(x) = 09x^4 + 87x^3 + 65x^2 + 43x + 21,$$

$$B(x) = \sum_{i=0}^{3-1} B_i x^i = B_2 x^2 + B_1 x + B_0 = 09x^2 + 87x + 65,$$

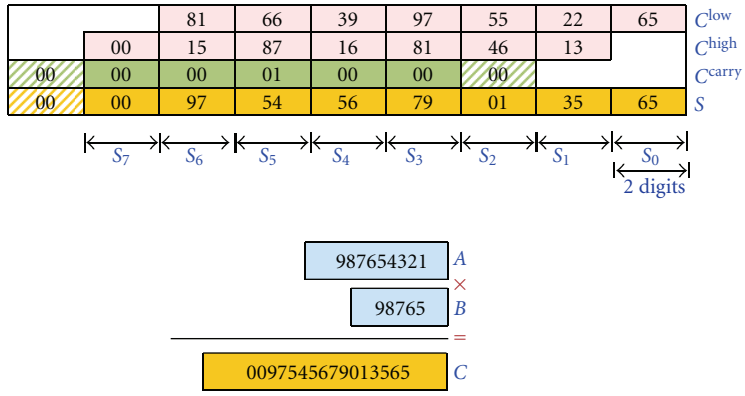
$$C(x) = 81x^6 + 1566x^5 + 8739x^4 + 11697x^3 + 8155x^2 + 4622x + 1365. \tag{7}$$

As described in Section 4.2, multiplication is performed as a two-step CAME process, as shown in Figure 5. The MAC process is first applied to calculate the convolution of the two numbers, see Figure 5(a), after which the partial results are then merged, as shown in Figure 5(b), to obtain the final result.

*4.4. Precision of the Arithmetic Unit.* One challenge of implementing an arbitrary-precision arithmetic unit is to ensure that it is possible to operate with any realistic size of operands. It is therefore necessary to investigate the growth of the MAC process as it represents an upper bound for the unit precision. As discussed earlier, shown in Figure 2, and given by (4c), the outcome of the MAC process, $C_i$, consists of three parts: the multiply digits, $Q_i^{low}$, $Q_i^{high}$, and the accumulation digits, $\Delta_i$. The corresponding number of digits,




(a) Multiply-accumulate (MAC) process







(b) Merge process

Figure 5: Example of decimal multiplication.

$n_{C_i}$, $n_{Q_i^{low}}$, $n_{Q_i^{high}}$, $n_{\Delta_i}$, can be expressed as given by (8a), (8b), (8c), and shown in Figure 6:

$$n_{C_i} = n_{Q_i^{low}} + n_{Q_i^{high}} + n_{\Delta_i}, \qquad n_{Q_i^{low}} = n_{Q_i^{high}} = p, \tag{8a}$$

$$n_{\Delta_i} = \begin{cases} \log_r(i+1), & 0 \le i \le (n_2 - 2), \\ \log_r(n_2), & (n_2 - 1) \le i \le (n_1 - 1), \\ \log_r(n_1 + n_2 - 1 - i), & n_1 \le i \le (n_1 + n_2 - 2), \end{cases} \tag{8b}$$

$$n_{\Delta_{max}} = \left\lceil \log_r\left(\frac{m_2}{p}\right)\right\rceil,$$

$$0 \le n_{\Delta_{max}} \le p \Longrightarrow p \le m_2 \le p \cdot r^p,$$

when

$$p = 64 \Longrightarrow 64 \le m_2 \le 64 \cdot 2^{64} \equiv 128 \text{ EiB (ExibiByte)}. \tag{8c}$$

Controlling the growth of the MAC process by keeping the accumulation/carry digits less than or equal to $p$-digits, (8c) gives the upper bound on the precision of the input operands. This is in terms of the hardware unit word-length, that is, $p$-digits, and the numeric system base $r$. For example, for a binary representation, that is, $r = 2$, and a 64-bit arithmetic unit, that is, $p = 64$, the accommodated operand precision is 128 ExibiByte which is beyond any realistic storage system. In other words, a hardware unit with such parameters can provide almost infinite-precision arithmetic.

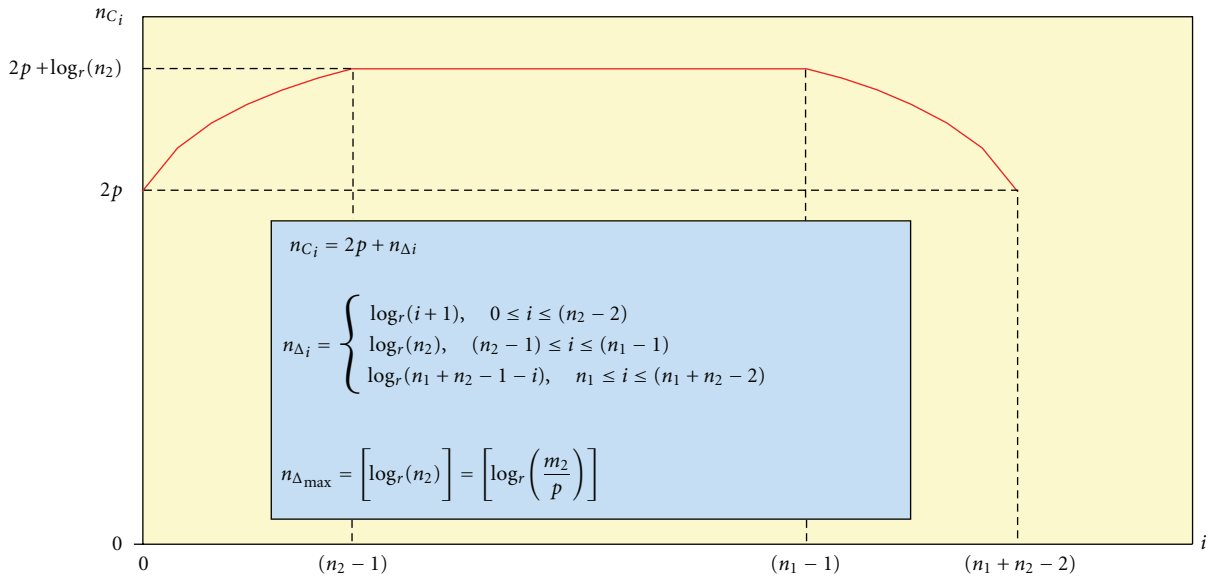



Figure 6: Growth of MAC process (accumulation/carry digits).

## 5. Experimental Work

Our experiments have been performed on one representative HPRC systems, SRC-6 [17], previously described in Section 2. The proposed architecture was developed partly in Xilinx System Generator environment as well as in VHDL. In both environments, the architectures were highly parameterized.

The hardware performance was referenced to one of the most efficient [21] software libraries supporting arbitrary-precision arithmetic, namely, GMP library on Xeon 2.8 GHz. We considered two versions of GMP. The first was the compiled version of GMP which is a precompiled and highly optimized version for the underlying microprocessor. The other was the highly portable version of GMP without any processor-specific optimizations.

*5.1. Implementation Issues.* Large-precision reduction operations used in both the MAC (i.e., $3p$-digit accumulation) and the merging processes proved to be a challenge due to critical-path issues. For example, the accumulation of a stream of integers can be impractical for FPGA-based implementations when the number of values is large. The resultant circuit can significantly reduce the performance and consume an important portion of the FPGA.

To eliminate those effects of reduction operations, techniques of deep pipelining [25, 26] and those of nonlinear pipelining [27, 28] were considered. The buffering mechanism, presented in [25, 26], showed either low throughput and efficiency, or high latency and resources usage for our case, see Figure 7(a). Therefore, we leveraged the techniques of nonlinear pipelines [27, 28] which proved to be effective, see Figure 7(b). Furthermore, we derived a generalized architecture for nonlinear pipelined accumulation; refer to (9). The main component of this structure is a $p$-digit accumulation stage in which delay elements are added for synchronization purposes, see Figure 8. The $p$-digit stages are arranged in a manner such that overflow digits from a given stage $j$ are passed to the subsequent stage $j + 1$. The total number of stages $n_s$ depends on the size of the input operand $m_A$ as well as on the number of accumulated operands $N$, see (9). In doing so, we have proved

$$S = \sum_{i=0}^{N-1} A_i, \qquad A_i = \sum_{j=0}^{n_A-1} A_{i,j} x^j,$$

$$S = \sum_{i=0}^{N-1} \left( \sum_{j=0}^{n_A-1} A_{i,j} x^j \right) = \sum_{j=0}^{n_A-1} \left( \sum_{i=0}^{N-1} A_{i,j} x^j \right)$$

$$= \sum_{j=0}^{n_A-1} \left( x^j \sum_{i=0}^{N-1} A_{i,j} \right),$$

$$S = \sum_{j=0}^{n_A-1} S_{N-1,j} x^j,$$

$$S_{N-1,j} = \sum_{i=0}^{N-1} A_{i,j} \iff S_{i,j} = S_{i-1,j} + A_{i,j},$$

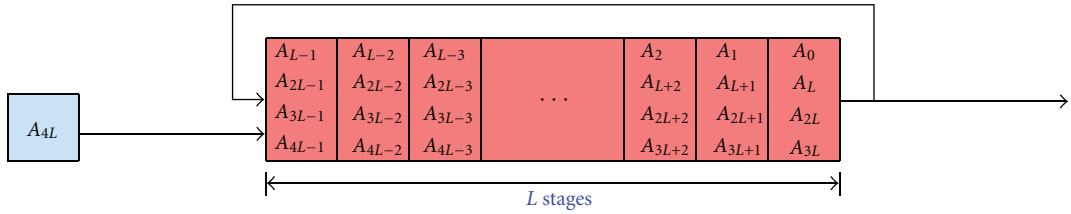
$$\forall i \in [0, N-1], \quad S_{-1,j} = 0,$$

$$S_{i,j} = Q_{i,j} + \Delta_{i,j} \cdot x,$$

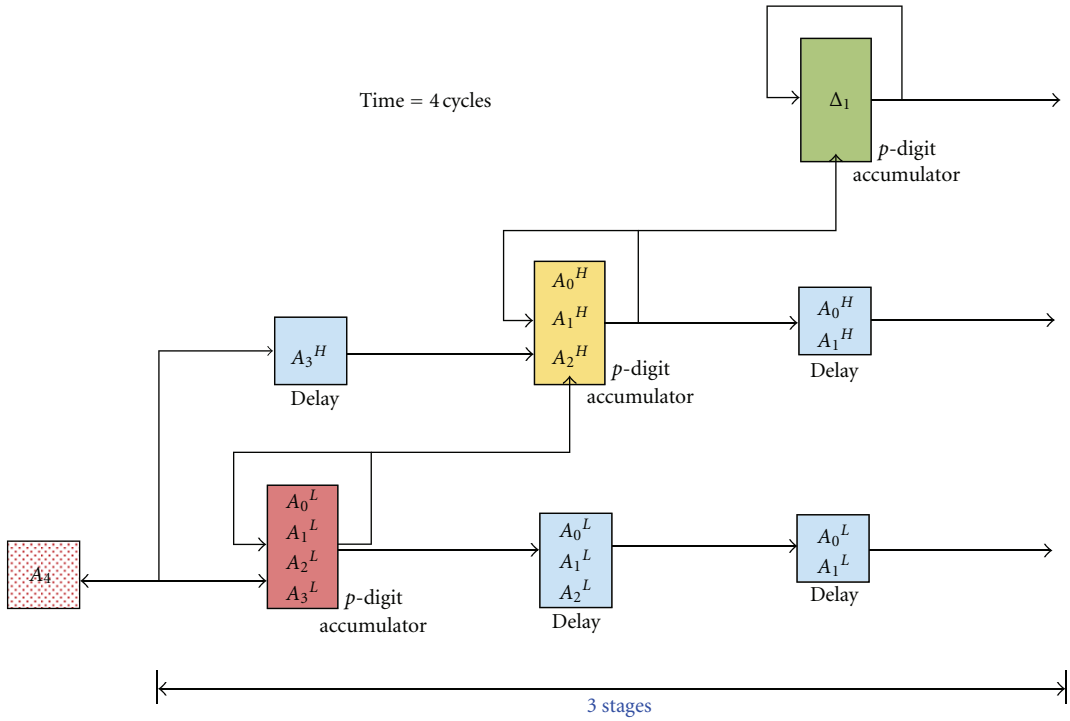
$$Q_{i,j} + \Delta_{i,j} \cdot x = Q_{i-1,j} + \Delta_{i-1,j} \cdot x + A_{i,j},$$

$$Q_{i-1,j} + A_{i,j} = Q_{i,j} + c_{i,j} \cdot x,$$

$$\Longrightarrow \Delta_{i,j} = \Delta_{i-1,j} + c_{i,j} \iff \Delta_{N-1,j} = \sum_{i=0}^{N-1} c_{i,j},$$




(a) Deeply pipelined accumulator




(b) Proposed nonlinear-pipelined accumulator

Figure 7: Accumulator of the MAC process.

where $x = r^p$, $m_S = m_A + m_\Delta$, $m_\Delta = \log_r N$,

$$n_A = \left\lceil \frac{m_A}{p} \right\rceil, \quad n_\Delta = \left\lceil \frac{m_\Delta}{p} \right\rceil,$$

$$n_S = n_A + n_\Delta = \left\lceil \frac{m_A}{p} \right\rceil + \left\lceil \frac{\log_r N}{p} \right\rceil, \tag{9}$$

that large-size accumulation operations which are prone to deeply-pipelined effects and hardware critical-path issues can be substituted with multiple and faster smaller-size accumulations. In other words, a single large-size $(p \cdot n_s)$-digit accumulator can efficiently be implemented using $n_s$ faster $p$-digit accumulators, see Figure 8.

We analyzed the pipeline efficiency, as defined in [27, 28], of our proposed architecture. This is expressed by (10a), (10b), and shown in Figure 10. The pipeline efficiency $\eta$ as expressed by (10a) can be easily derived by considering the pipeline reservation table [27, 28]. As shown in Figure 9, the example reservation table is used to calculate the pipeline efficiency $\eta$. We implemented two versions of the arithmetic unit, that is, 32-bit and 64-bit. For very large data, the efficiency for the 32-bit unit was lower bounded to 80% while the efficiency for the 64-bit unit was lower bounded to 84.62%, see (10b) and Figure 10:

$$\eta = \frac{\text{Full Cells}}{\text{Total Cells}} = 1 - \frac{\text{Empty Cells}}{\text{Total Cells}},$$

$$\eta = 1 - \frac{L_{\text{merger}}(n_1 - 1)(n_2 - 1)}{L_{\text{total}} n_1 n_2}$$

$$= 1 - \frac{1}{1 + L_{\text{mac}}/L_{\text{merger}}} \cdot \left(1 - \frac{1}{n_1}\right)\left(1 - \frac{1}{n_2}\right),$$

where $L_{\text{mac}} \equiv$ Latency of the MAC − process,

$L_{\text{merger}} \equiv$ Latency of the merging − process,

$L_{\text{total}} = L_{\text{mac}} + L_{\text{merger}}$, (10a)

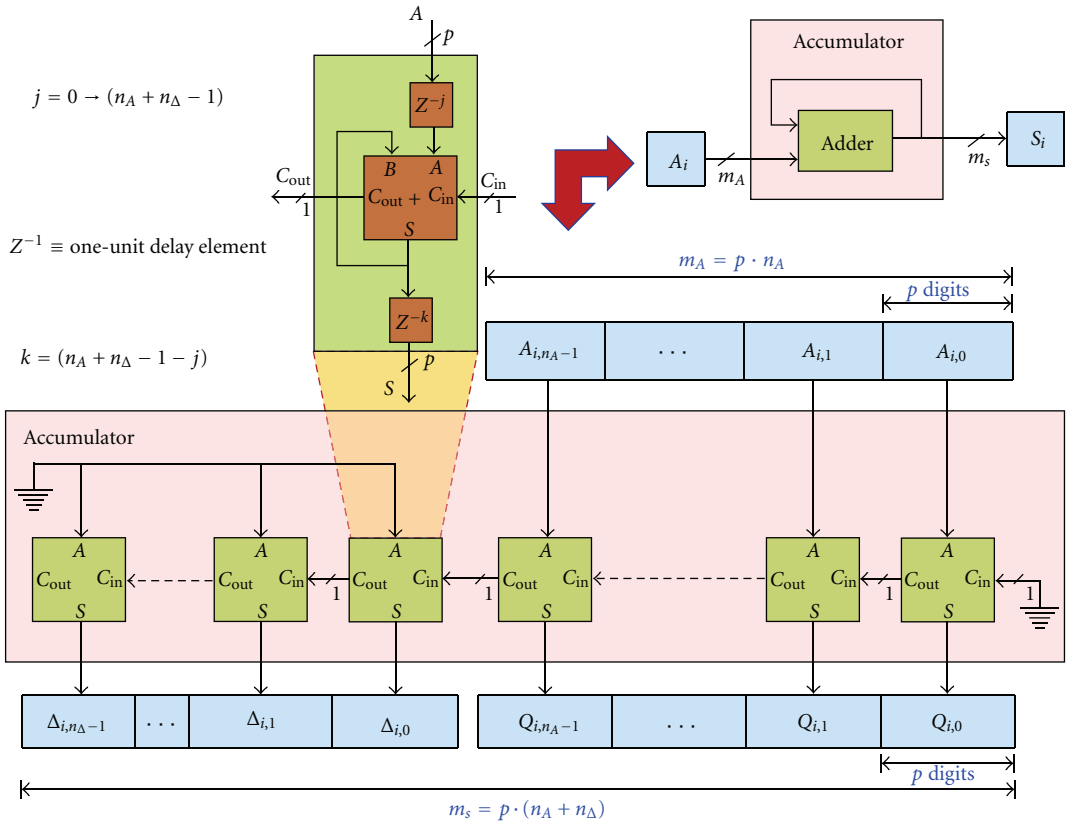
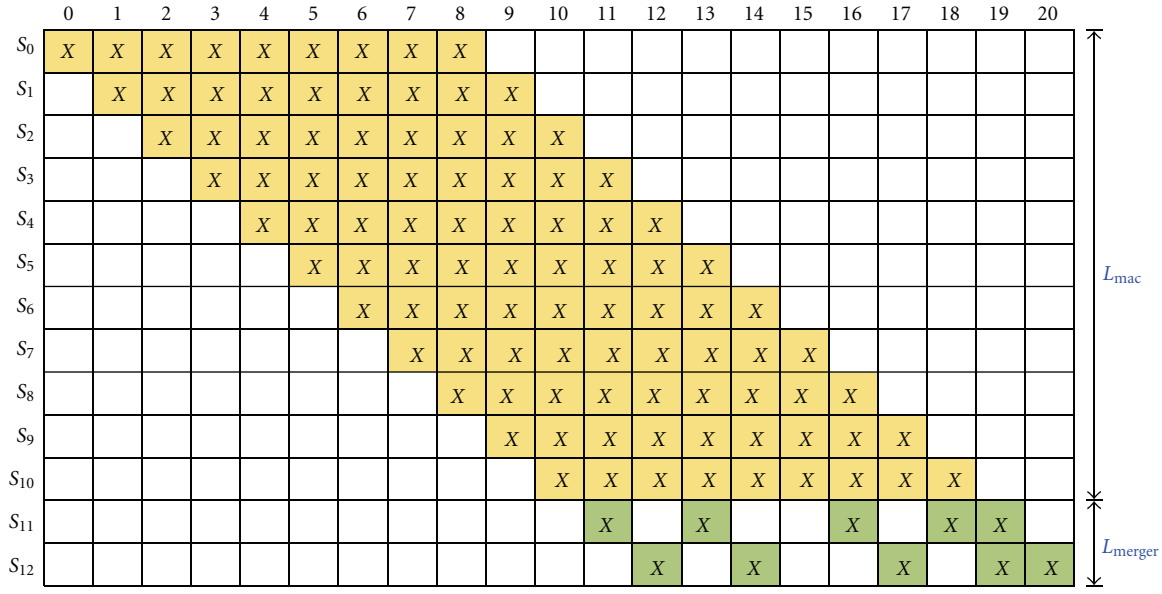



Figure 8: Generalized nonlinear-pipelined accumulator.




Data size = $3p$ bits (i.e., $n_1 = n_2 = 3$)

$$\eta = \frac{\text{Full cells}}{\text{Total cells}} = 1 - \frac{\text{Empty cells}}{\text{Total cells}} = 1 - \frac{2 \times 4}{13 \times 9} = 93.16\%$$

Figure 9: Example pipeline reservation table.

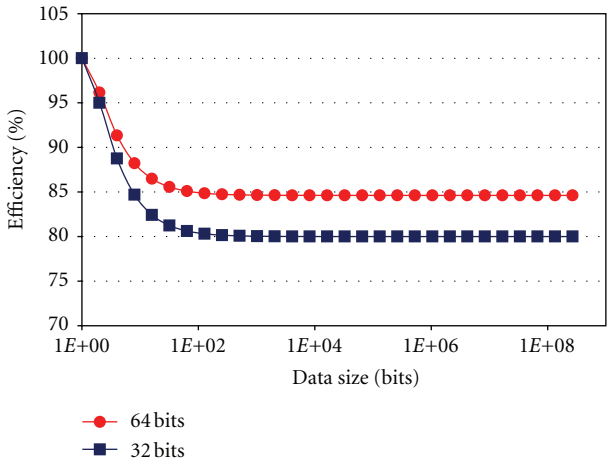



Figure 10: Pipeline efficiency.

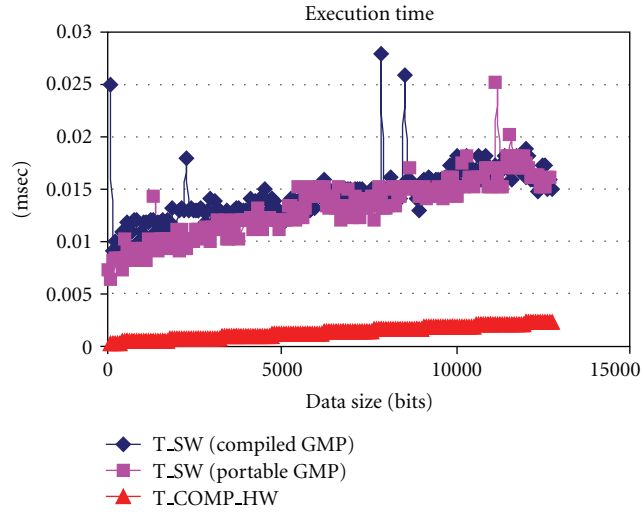



Figure 11: Addition/subtraction execution time (64-bit).

Table 2: FPGA resource utilization.

| | 32-bit unit | 64-bit unit |
|---|---|---|
| Slice flip flops | 6,155 (9%) | 9,183 (13%) |
| 4-input LUTs | 2,524 (3%) | 6,311 (9%) |
| Occupied slices | 3,983 (11%) | 6,550 (19%) |
| 18 × 18b multipliers | 4 (2%) | 10 (6%) |
| Clock frequency | 102.8 MHz | 100.4 MHz |

$$\eta_\infty = \lim_{n_1, n_2 \to \infty} \eta = \frac{L_{\text{mac}}}{L_{\text{total}}},$$

when

$$p = 32\,\text{bits}, \quad L_{\text{mac}} = 4, \quad L_{\text{merger}} = 1 \Longrightarrow \eta_\infty = 80.00\%,$$
$$p = 64\,\text{bits}, \quad L_{\text{mac}} = 11, \quad L_{\text{merger}} = 2 \Longrightarrow \eta_\infty = 84.62\%. \tag{10b}$$

*5.2. Experimental Results.* Our experiments were performed for two cases, that is, 32-bit and 64-bit units. FPGA resources usage and clock frequency are shown in Table 2. While resource usage in the 64-bit unit is larger, as expected, clock frequency is similar in both cases due to the clock frequency requirement imposed by SRC-6 (100 MHz). As it can be seen in Table 2, the proposed architecture consumes relatively low hardware resources requiring at maximum 19% of the reconfigurable device for the 64-bit unit. These implementation results allow taking advantage of the inherent parallelism of the FPGA and adding more than one operational unit upper bounded by the number of available memory banks. For example, in SRC-6, there are six memory banks which make it possible to allocate two units per FPGA, see Figure 4(a).

Next, we show the results of the 64-bit Basecase algorithm for addition/subtraction, and the 32-bit and 64-bit Basecase algorithm for multiplication.

The arbitrary-precision addition/subtraction performance was measured on SRC-6 and compared to both

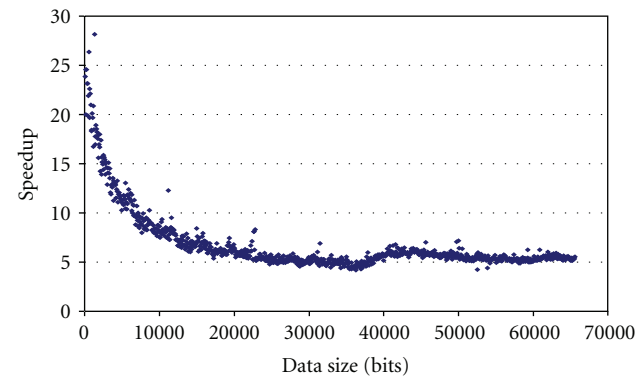
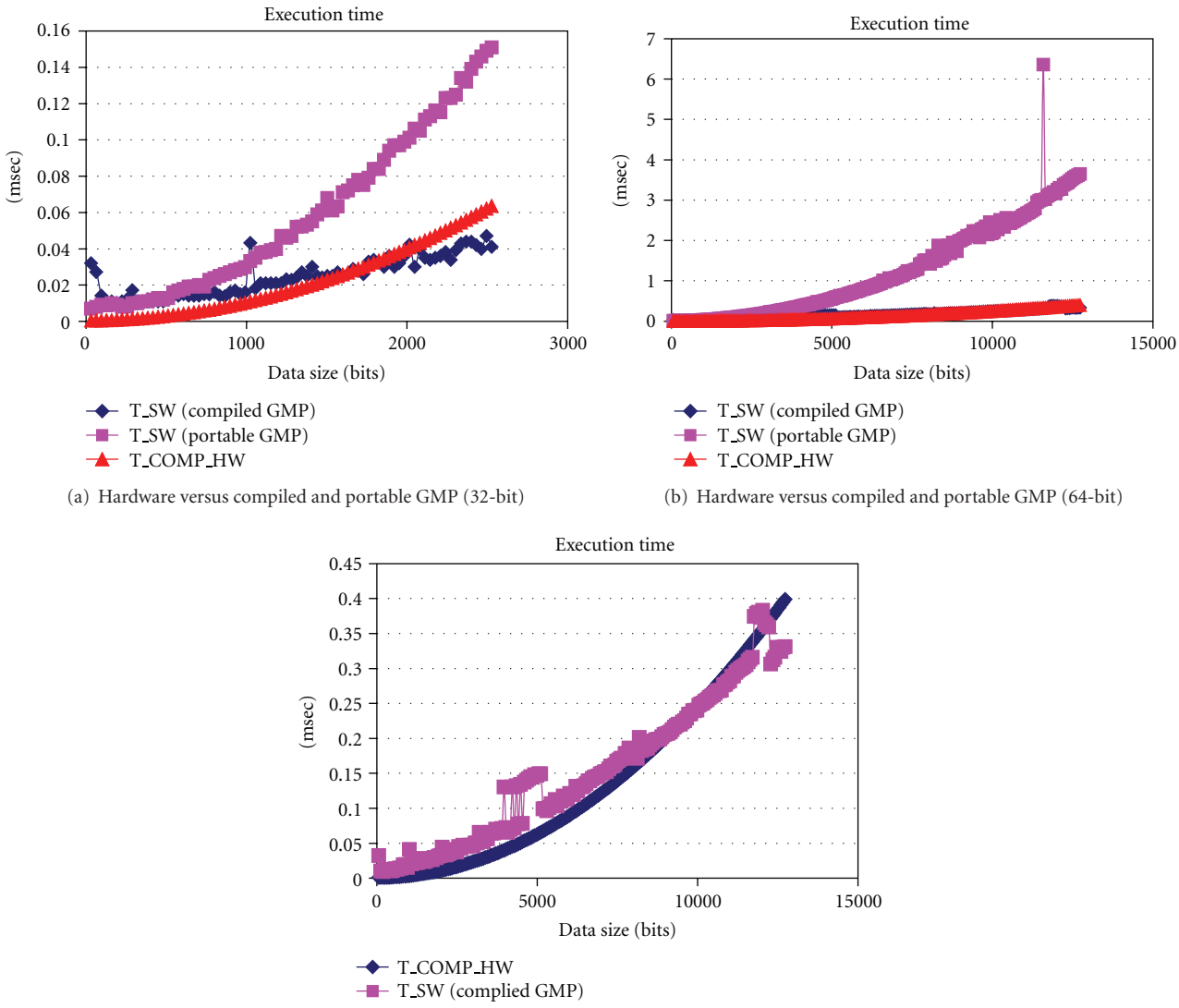



Figure 12: Addition/subtraction hardware speedup versus GMP (64-bit).

the compiled and portable versions of GMP. As shown in Figure 11, T_COMP_HW, that is, the total computation time of the hardware on SRC-6, is lower than the execution time of both the compiled and portable versions of GMP. The performance speedup is shown in Figure 12. The hardware implementation asymptotically outperforms the software, by a factor of approximately 5, because of the inherent parallelism exploited by the hardware. We can also notice that, for small-precision addition/subtraction, the speedup factor starts from approximately 25. This is due to the large overhead, relative to the data size, associated with the software, while the only overhead associated with the hardware is due to the pipeline latency. This latency is independent on the data size. It is also worth to notice the linear behavior, $O(n)$, of both the software and the hardware. This is because both execute the same algorithm, that is, Bascase addition/subtraction [20, 21], see Table 1.

In the case of multiplication, we notice a nonlinear behavior $O(n^{1+e})$, $0 < e < 1$; see Figure 13 and Table 1. We notice also a similar behavior to the addition/subtraction for small-size operands. Figures 13(a) and 13(b) show a significant performance for the hardware compared to the portable




(a) Hardware versus compiled and portable GMP (32-bit)

(b) Hardware versus compiled and portable GMP (64-bit)

(c) Hardware versus compiled GMP (64-bit)

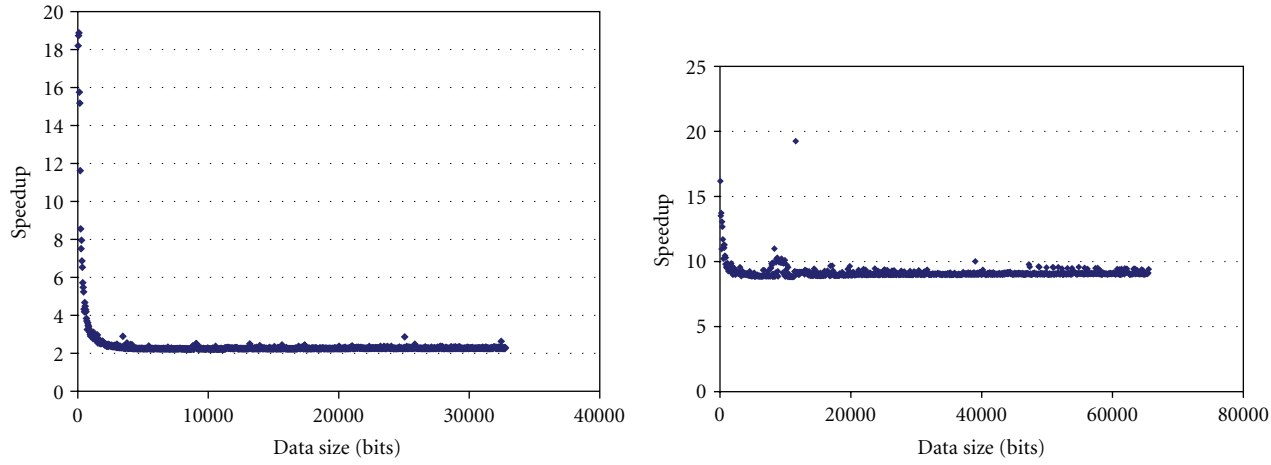
Figure 13: Multiplication execution time.

version of GMP. This is because this version of GMP uses the same algorithm as ours, that is, Basecase with $O(n^2)$ see Table 1, independent of the data size [20, 21]. As shown in Figure 14, the hardware behavior asymptotically outperforms the portable GMP multiplication by a factor of approximately 2 for the 32-bit multiplication, see Figure 14(a), and 9 for the 64-bit multiplication, see Figure 14(b). However, this is not the case with the compiled GMP multiplication which is highly optimized and adaptive. Compiled GMP uses four multiplication algorithms, and adaptively switches from a slower to a faster algorithm depending on the data size and according to predetermined thresholds [20, 21]. For these reasons, the hardware, as can be seen from Figure 13(c), outperforms the compiled GMP up to a certain threshold, approximately 10 Kbits, beyond which the situation reverses.

## 6. Conclusions and Future Work

This paper shows the feasibility of accelerating arbitrary-precision arithmetic on HPRC platforms. While exact computation presents many benefits in terms of numerical robustness, its main drawback is the poor performance that is obtained in comparison to fixed-precision arithmetic. The results presented in this work show the possibility of reducing the performance gap between fixed-precision and arbitrary-precision arithmetic using HPRC machines.

The proposed solution, the Convolve-And-MErge (CAME) methodology for arbitrary-precision arithmetic, is derived from a formal representation of the problem and is based on virtual convolution scheduling. For the formal analysis, only the multiplication operation was considered. This decision was made due to the fact that multiplication




(a) 32-bit (b) 64-bit

Figure 14: Multiplication hardware speedup versus portable GMP.

is a core operation from which other basic arithmetic operations, such as division and square root, can be easily derived.

Our proposed arithmetic unit can perform arbitrary-precision addition, subtraction, multiplication, as well as arbitrary-length convolution operations. Our approach in implementing the CAME process was based on leveraging previous work and concepts that were introduced for solving similar problems. Dynamic (nonlinear) pipelines techniques were exploited to eliminate the effects of deeply pipelined reduction operators. The use of these techniques allowed us reaching a minimum of 80% pipeline utilization for 32-bit units and reaching 84.6% efficiency for 64-bit units. This implementation was verified for both correctness and performance in reference to the GMP library on the SRC-6 HPRC. The hardware outperformed GMP by a factor of 5x speedup for addition/subtraction, while the speedup factor was lower bounded to 9x compared to the portable version of GMP multiplication.

Future directions may include investigating hardware support for floating-point arbitrary precision, considering faster algorithms than the Basecase/Schoolbook presented in this paper, as well as adopting methods for adaptive algorithm switching based on the length of the operands. In addition to, full porting of an arbitrary-precision arithmetic library such as GMP to HPRC machines might also be favorable.

## References

[1] V. Sharma, *Complexity analysis of algorithms in algebraic computation*, Ph.D. dissertation, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 2007.

[2] C. K. Yap and T. Dube, "The exact computation paradigm," in *Computing in Euclidean Geometry*, D. Z. Du and F. K. Hwang, Eds., vol. 4 of *Lecture Notes Series on Computing*, pp. 452–492, World Scientific Press, Singapore, 2nd edition, 1995.

[3] D. E. Knuth, "The art of computer programming," in *Seminumerical Algorithms*, vol. 2, Addison-Wesley, 3rd edition, 1998.

[4] http://en.wikipedia.org/wiki/Arbitrary_precision_arithmetic.

[5] C. Li, *Exact geometric computation: theory and applications*, Ph.D. dissertation, Department of Computer Science, Institute of Mathematical Sciences, New York University, 2001.

[6] http://bitsavers.org/pdf/ibm/140x/A24-1401-1_1401_System_Summary_Sep64.pdf.

[7] http://ibm-1401.info/1401-Competition.html#IntroHoneywell200.

[8] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (Revisiting Iterative Refinement for Linear Systems)," in *Proceedings of the ACM/IEEE SC Conference*, Tampa, Fla, USA, November 2006.

[9] J. Hormigo, J. Villalba, and E. L. Zapata, "CORDIC processor for variable-precision interval arithmetic," *Journal of VLSI Signal Processing Systems*, vol. 37, no. 1, pp. 21–39, 2004.

[10] S. Balakrishnan and S. K. Nandy, "Arbitrary precision arithmetic—SIMD style," in *Proceedings of the 11th International Conference on VLSI Design: VLSI for Signal Processing*, p. 128, 1998.

[11] A. Saha and R. Krishnamurthy, "Design and FPGA implementation of efficient integer arithmetic algorithms," in *Proceedings of the IEEE Southeastcon '93*, vol. 4, no. 7, April 1993.

[12] D. M. Chiarulli, W. G. Rudd, and D. A. Buell, "DRAFT—a dynamically reconfigurable processor for integer arithmetic," in *Proceedings of the 7th Symposium on Computer Arithmetic*, pp. 309–321, IEEE Computer Society Press, 1989.

[13] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *IEEE Computer*, vol. 41, no. 2, pp. 69–76, 2008.

[14] A. Michalski, D. Buell, and K. Gaj, "High-throughput reconfigurable computing: design and implementation of an idea encryption cryptosystem on the SRC-6e reconfigurable computer," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 681–686, August 2005.

[15] E. El-Araby, T. El-Ghazawi, J. Le Moigne, and K. Gaj, "Wavelet spectral dimension reduction of hyperspectral imagery on

a reconfigurable computer," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 399–402, Brisbane, Australia, December 2004.

[16] E. El-Araby, M. Taher, T. El-Ghazawi, and J. Le Moigne, "Prototyping Automatic Cloud Cover Assessment (ACCA) algorithm for remote sensing on-board processing on a reconfigurable computer," in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT '05)*, pp. 207–214, Singapore, December 2005.

[17] SRC Computers, *SRC Carte C Programming Environment v2.2 Guide (SRC-007-18)*, 2006.

[18] http://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations.

[19] T. Granlund and P. L. Montgomery, "Division by invariant integers using multiplication," in *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI '94)*, pp. 61–72, June 1994.

[20] GMP Manual, *GNU MP The GNU Multiple Precision Arithmetic Library*, 4.2.1 edition, 2006.

[21] http://gmplib.org/.

[22] H. L. Tredennick and T. A. Welch, "High-speed buffering for variable length operands," *Proceedings of the 4th Annual Symposium on Computer Architecture (ISCA '77)*, vol. 5, no. 7, pp. 205–210, March 1977.

[23] S. Olariu, M. C. Pinotti, and S. Q. Zheng, "How to sort N items using a sorting network of fixed I/O size," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 5, pp. 487–499, 1999.

[24] H. ElGindy and G. Ferizis, "Mapping basic recursive structures to runtime reconfigurable hardware," in *Proceedings of the FPL*, August 2004.

[25] L. Zhou, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 10, pp. 1377–1392, 2007.

[26] L. Zhuo and V. K. Prasanna, "High-performance and area-efficient reduction circuits on FPGAs," in *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '05)*, pp. 52–59, Rio de Janeiro, Brazil, October 2005.

[27] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGrawHill, 1993.

[28] K. Hwang and Z. Xu, *Scalable Parallel Computing: Technology, Architecture, Programming*, McGrawHill, 1998.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 752910, 15 pages
doi:10.1155/2012/752910

*Review Article*

# High Performance Biological Pairwise Sequence Alignment: FPGA versus GPU versus Cell BE versus GPP

**Khaled Benkrid,[1] Ali Akoglu,[2] Cheng Ling,[1] Yang Song,[2] Ying Liu,[1] and Xiang Tian[1]**

[1] *Institute of Integrated Systems, School of Engineering, The University of Edinburgh, Kings Buildings, Mayfield Road, Edinburgh EH9 3JL, UK*
[2] *Electrical and Computer Engineering Department, The University of Arizona, Tucson, AZ 85721-0104, USA*

Correspondence should be addressed to Khaled Benkrid, k.benkrid@ed.ac.uk

Received 15 December 2011; Revised 13 February 2012; Accepted 17 February 2012

Academic Editor: Kentaro Sano



This paper explores the pros and cons of reconfigurable computing in the form of FPGAs for high performance efficient computing. In particular, the paper presents the results of a comparative study between three different acceleration technologies, namely, Field Programmable Gate Arrays (FPGAs), Graphics Processor Units (GPUs), and IBM's Cell Broadband Engine (Cell BE), in the design and implementation of the widely-used Smith-Waterman pairwise sequence alignment algorithm, with general purpose processors as a base reference implementation. Comparison criteria include speed, energy consumption, and purchase and development costs. The study shows that FPGAs largely outperform all other implementation platforms on performance per watt criterion and perform better than all other platforms on performance per dollar criterion, although by a much smaller margin. Cell BE and GPU come second and third, respectively, on both performance per watt and performance per dollar criteria. In general, in order to outperform other technologies on performance per dollar criterion (using currently available hardware and development tools), FPGAs need to achieve at least two orders of magnitude speed-up compared to general-purpose processors and one order of magnitude speed-up compared to domain-specific technologies such as GPUs.

## 1. Introduction

Since it was first announced in 1965, Moore's law has stood up the test of time, providing exponential increases in computing power for science and engineering problems over time. However, while this law was largely followed through increases in transistor integration levels and clock frequencies, this is no longer possible as power consumption and heat dissipation are becoming major hurdles in the face of further clock frequency increases, the so-called frequency or power wall problem.

In order to keep Moore's law going, general-purpose processor manufacturers, for example, Intel and AMD, are now relying on multicore chip technology in which multiple cores run simultaneously on the same chip at capped clock frequencies to limit power consumption. While this has the potential to provide considerable speed-up for science and engineering applications, it is also creating a semantic gap between applications, traditionally written in sequential code, and hardware, as multicore technologies need to be programmed in parallel to take advantage of their performance potential. This problem is however also opening a window of opportunity for hitherto niche parallel computer technologies such as Field Programmable Gate Arrays (FPGAs) and Graphics Processor Units (GPUs) since the problem of parallel programming has to be tackled for general-purpose processors anyway.

This paper presents a comparative study between three different acceleration technologies, namely, Field Programmable Gate Arrays (FPGAs), Graphics Processor Units (GPUs), and IBM's Cell Broadband Engine (Cell BE), in the design and implementation of the widely-used Smith-Waterman pairwise sequence alignment algorithm, with general purpose processors as a base reference implementation. Comparison criteria include the speed of the resulting implementation, its energy consumption, as well as purchase

and development costs. Note that the aim of this paper is not to present the best implementation (from a speed point of view) on the four architectures but to perform a fair comparison of all four technologies in terms of speed, energy consumption, and development time and cost. We thus chose not to use the results of the best implementations reported in the literature, but instead to perform our own experiments using a set of Ph.D. students with relatively equal experience on each platform and measure the speed, development time, cost and energy consumption of each resulting implementation.

The rest of this paper is organized as follows. The following section will first present background on the Smith-Waterman algorithm, together with an overview of the target implementation platforms, namely, Xilinx Virtex-4 FPGAs, NVIDIA GeForce 8800GTX GPU, IBM's Cell BE processor and finally the Pentium 4 Prescott processor. Sections 3, 4, 5, and 6 will then report our design and implementation of the Smith-Waterman algorithm on each of the above platforms, in turn. After that, comparative implementation results on all platforms are presented in Section 7 before final conclusions are drawn.

## 2. Background

Pairwise biological sequence alignment is a basic operation in the field of bioinformatics and computational biology with a wide range of applications in disease diagnosis, drug engineering, biomaterial engineering, and genetic engineering of plants and animals [1]. The aim of this operation is to assign a score to the degree of similarity or correlation between two sequences, for example, Protein or DNA, which can then be used to find out whether two sequences are related or not, build a multiple sequence alignment profile, or construct phylogenetic trees. The most accurate algorithms for pairwise sequence alignment are exhaustive search dynamic-programming- (DP-) based algorithms such as the Needleman-Wunsch algorithm [2] and the Smith-Waterman algorithm [3]. The latter is the most commonly used DP algorithm as it finds the best local alignment of subsegments of a pair of biological sequences. However, biological sequence alignment is also a computationally expensive application as its computing and memory requirements grow quadratically with the sequence length [4]. Given that a query sequence is often aligned to a whole database of sequences in order to find the closest matching sequence (see Figure 1) and given the annual increase in the size of biological databases, there is a need for a matching increase in computing power at reasonable cost [5].

The following subsections will present theoretical background on the Smith-Waterman algorithm, followed by an architectural overview of each of the four target hardware platforms.

*2.1. The Smith-Waterman Algorithm for Pairwise Biological Sequence Alignment.* Biological sequences, for example, DNA or protein sequences of residues (a DNA residue is one of four nucleotides while a protein residue is one of 20 aminoacids [1]) evolve through a process of mutation, selection, and random genetic drift [6]. Mutation, in particular, manifests itself through three main processes, namely, *substitution* of residues (i.e., a residue A in the sequence is replaced by another residue B), *insertion* of new residues, and *deletion* of existing residues. Insertion and deletion are referred to as *gaps*. The gap character "-" is introduced to present a character insertion or deletion between sequences. There are four ways to indicate the alignment between two sequence $s$ and $t$ as shown below:

$(a, a)$ denotes a match (no change from $s$ to $t$),

$(a, -)$ denotes deletion of character $a$ (in $s$),

$(a, b)$ denotes replacement of $a$ (in $s$) by $b$ (in $t$),

$(-, b)$ denotes insertion of character $b$ (in $s$).

For example, an alignment of two sequences $s$ and $t$ (see Table 1) is an arrangement of $s$ and $t$ by position, where $s$ and $t$ can be padded with gap symbols to achieve the same length and where $(A, A)$ indicates a match, $(G, -)$ indicates the deletion of $G$, $(-, T)$ indicates the insertion of $T$, and $(C, A)$ indicates the replacement of $C$ by $A$.

The most basic pairwise sequence analysis task is to ask whether two sequences are related or not, and by how much. It is usually done by first aligning the sequences (or part of sequences) and then deciding whether the alignment is more likely to have occurred because the sequences are related or just by chance. The parameters of the alignment methods are [1] as follows:

(i) the types of alignment to be considered;

(ii) the scoring system used to rank the alignments;

(iii) the algorithm used to find optimal (or good) scoring alignments;

(iv) the statistical methods used to evaluate the significance of an alignment score.

The degree of similarity between pairs of biological sequences is measured by a score, which is a summation of odd-log scores between pairwise residues, in addition to gap penalties. The odd-log scores are based on the statistical

Table 1: Denotations of the alignment between sequences $s$ and $t$.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $s$: | $A$ | $G$ | $C$ | $A$ | $C$ | $A$ | $C$ | $-$ | $C$ |
| $t$: | $A$ | $-$ | $C$ | $A$ | $C$ | $A$ | $C$ | $T$ | $A$ |

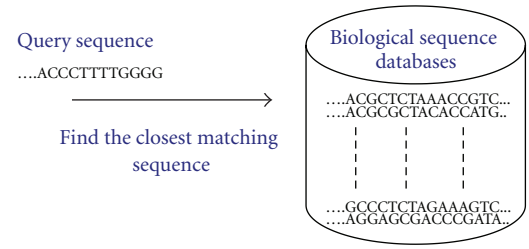



Figure 1: Pairwise sequence alignment, for example, DNA.

likelihood of any possible alignment of pairwise residues and is often summarised in a substitution matrix (e.g., BLOSUM50, BLOSUM62, PAM). The gap penalty depends on the length of gaps and is often assumed independent of the gap residues. There are two types of gap penalties, known as *linear gaps* and *affine gaps*. The linear gap is a simple model with constant gap penalty ($d$) multiplied by the length of the gap ($g$), denoted as

$$\text{Penalty}\ (g) = -g * d. \tag{1}$$

An Affine gap has opening and extension penalties. The constant penalty for opening a gap is normally bigger than the penalty for extending a gap, which is more biologically realistic as few gaps are as frequent as a single gap in practice. Affine gaps are thus formulated as ($d$ is the opening penalty and $e$ is the extension penalty)

$$\text{Penalty}\ (g) = -d - (g-1) * e, \quad \text{where } d > e. \tag{2}$$

For the sake of simplicity the following presents the Smith-Waterman algorithm in the case of linear gaps. The extension to the case of affine gaps is straightforward [1].

The Smith-Waterman (SW) algorithm is a widely used pairwise sequence alignment algorithm as it finds the best possible aligned subsegments in a pair of sequences (the so-called local alignment problem). It entails the construction of an alignment matrix ($F$) by a recursion equation as shown for an alignment between two sequences $X = \{x_i\}$ and $Y = \{y_j\}$:

$$F(i,j) = \max \begin{cases} 0, \\ F(i-1,j-1) + s\left(x_i, y_j\right), \\ F(i-1,j) - d, \\ F(i,j-1) - d. \end{cases} \tag{3}$$

Here, the alignment score is the largest of three alternatives (saturated to zero in case all three values are negative as it is better to start a new subsegment alignment than continue a subalignment with a negative score). These three alternatives are:

(i) An alignment between $x_i$ and $y_j$, in which case the new score is $F(i-1, j-1) + s(x_i, y_j)$, where $s(x_i, y_j)$ is the substitution matrix score or entry for residues $x_i$ and $y_j$.

(ii) An alignment between $x_i$ and a gap in $Y$, in which case the new score is $F(i-1, j) - d$, where $d$ is the gap penalty.

(iii) An alignment between $y_j$ and a gap in $X$, in which case the new score is $F(i, j-1) - d$, where $d$ is the gap penalty.

The dependency of each cell is shown in Figure 2. Here, each cell on the diagonal of the alignment matrix is independent of each other, which allows for a systolic architecture to be used in hardware in order to exploit this parallelism and hence speed up the algorithm execution.

After populating the alignment matrix, the best alignment between $X$ and $Y$ is obtained by tracing back from the cell with the maximum score in the alignment matrix back to the first zero matrix element. For this, we keep track of the matrix cell from which each cell's $F(i, j)$ was derived, that is, above, left, or above-left. The complete Smith-Waterman algorithm is illustrated in Figure 3 using the BLOSUM50 substitution matrix and a linear gap penalty equal to 8.

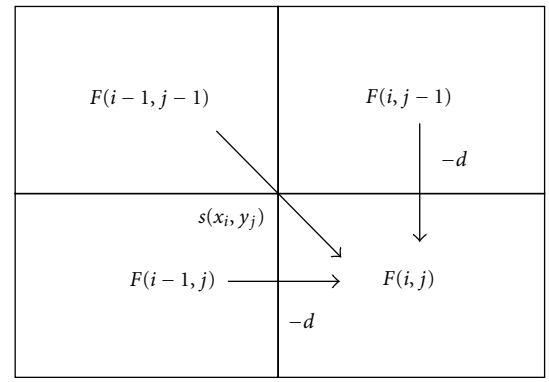



Figure 2: Data dependency of dynamic programming algorithms.

| | | H | E | A | G | A | W | G | H | E | E |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| P | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 0 | 5 | 0 | 5 | 0 | 0 | 0 | 0 | 0 |
| W | 0 | 0 | 0 | 0 | 2 | 0 | 20 | 12 | 4 | 0 | 0 |
| H | 0 | 10 | 2 | 0 | 0 | 0 | 12 | 18 | 22 | 14 | 6 |
| E | 0 | 2 | 16 | 8 | 0 | 0 | 4 | 10 | 18 | 28 | 20 |
| A | 0 | 0 | 8 | 21 | 13 | 5 | 0 | 4 | 10 | 20 | 27 |
| E | 0 | 0 | 6 | 13 | 18 | 12 | 4 | 0 | 4 | 16 | 26 |

Best local alignment: A W G H E
A W-H E

Query sequence: H E A G A W G H E E

Subject sequence: P A W H E A W

Figure 3: Illustration of the Smith-Waterman algorithm.

The following subsections will present an architectural overview of each of the four implementation platforms, in turn, namely, Xilinx' Virtex-4 FPGAs, NVIDIA's GeForce 8800GTX GPU, IBM's Cell BE processor, and Intel's Pentium 4 Prescott processor. To enable a fair comparison, these specific implementation platforms were chosen because they are all based on 90 nm CMOS technology and were purchased off-the-shelf at around the same time. Moreover, each platform was targeted by a different but equally experienced programmer.

*2.2. The FPGA Implementation Platform.* For the purpose of our FPGA-based implementation of the Smith-Waterman algorithm, we targeted an HP ProLiant DL145 server machine [7] which has an AMD 64bit processor and a Celoxica RCHTX FPGA board [8]. The latter has a Xilinx Virtex-4 LX160-11 FPGA chip, which is based on 90 nm

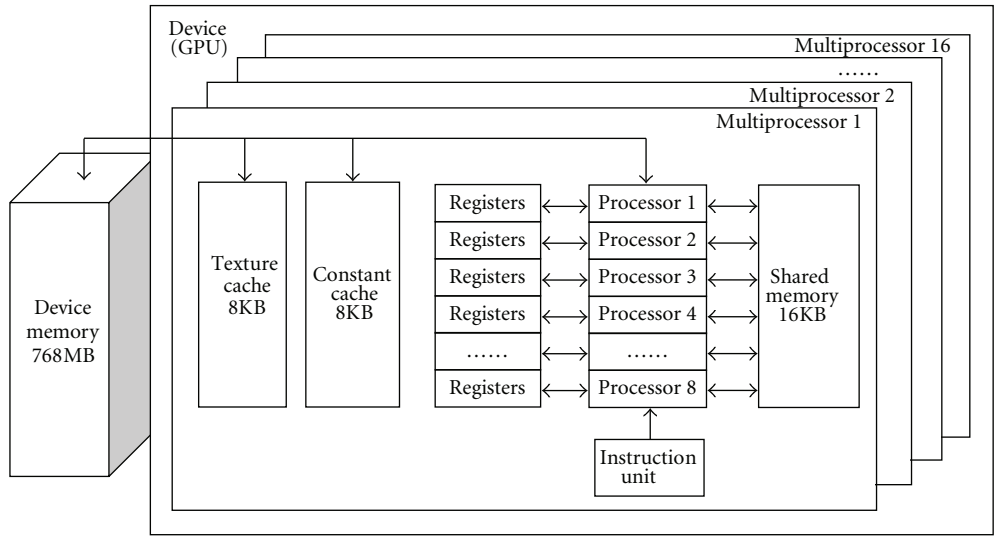



Figure 4: Architecture of NVIDIA's GeForce 8800 GTX.

copper CMOS process with a core voltage of 1.2 V [9]. All data transfer between the host processor and FPGA chip on the HP ProLiant server pass through the Hyper-Transport interface with a bandwidth of 3.2 GB/s.

The XC4VLX160 FPGA (see [9]) contains 67,584 slices, 1056 Kb of distributed memory, 96 XtremeDSP slices (not used in this paper's application) which can be configured as $18 \times 18$ multiplier with 48-bit accumulator, 288 BlockRAMs each 18 Kbit in size and configurable in dual ported mode with various word lengths and depths, and 960 user I/Os. Each slice has two 4-input look-up tables, which can be configured as $16 \times 1$ RAM, and two flip-flops in addition to some dedicated logic for fast addition and multiplication.

The FPGA design for the Smith-Waterman algorithm was captured in a C-based high level hardware language, called Handel-C [10], with the DK5 suite used to compile Handel-C into FPGA netlist, and Xilinx ISE software used for generating FPGA configuration bitstreams. A host application written in C++ services user queries and transfers them onto the FPGA board through the Hyper-Transport link. The FPGA configuration accepts a query sequence using an input/output interface based on the DSM library in Handel-C and starts alignment processing against a sequence database held on the FPGA board memory. Alignment results are then fed back to the host application through the FPGA input/output interface and Hyper-Transport link.

*2.3. The GPU Platform.* For the purpose of our GPU-based implementation of the Smith-Waterman algorithm, we targeted the GeForce 8800GTX GPU from NVIDIA Corp. [11]. This GPU is fabricated in 90 nm CMOS technology and consists of 16 Stream Multiprocessors (SMs), with each SM having eight Stream Processors (SPs) used as Arithmetic Logic Units (ALUs) with 8 KB constant cache, 8 KB texture cache, and 16 KB shared memory (see Figure 4). The SP clock frequency is 1,350 MHz.

This architecture, known as CUDA (Compute Unified Device Architecture), is a generic parallel computing architecture developed by NVIDIA Corp. to make the computing engines of graphics processing units accessible to general purpose software developers through a standard programming language, for example, C, with an API to exploit the architecture parallelism. Like many-core CPUs, CUDA uses threads for parallel execution. However, whereas multicore CPUs have only few threads running in parallel at any particular time, GPUs allow for thousands of parallel threads to run at the same time (768 threads per SM in the case of the GeForce 8800 GTX).

The memory hierarchy in CUDA devices consists of registers, shared memory, global memory, texture memory, and constant memory. Each SP has its own registers (1024) and operates the same kernel code as other SPs, but with different data sets. Shared memory (16 KB per SM) can be read and written to by any thread in a block of threads (or thread block) assigned to an SM. Access speed to shared memory is as fast as accessing SP registers as long as there are no bank conflicts [12]. Device memory offers global access to a larger (768 MB) but slower storage. Any thread in any SP can read from or write to any location in the global memory. Since computational results can be transferred back to CPU memory through it, global memory can be thought of as a bridge which achieves communication between GPU and CPU.

Local shared memory is allocated automatically if the size of variable required is bigger than the register size. It is not cached and cannot be accessed in a coalesced manner like global memory. Texture memory within each SM can be filled with data from the global memory. It acts as a cache, and so does constant memory, which means that their data fetch time is shorter. However, threads running in the SMs are restricted to read only access to these memories. The host CPU, on the other hand, does have write access to these memories.

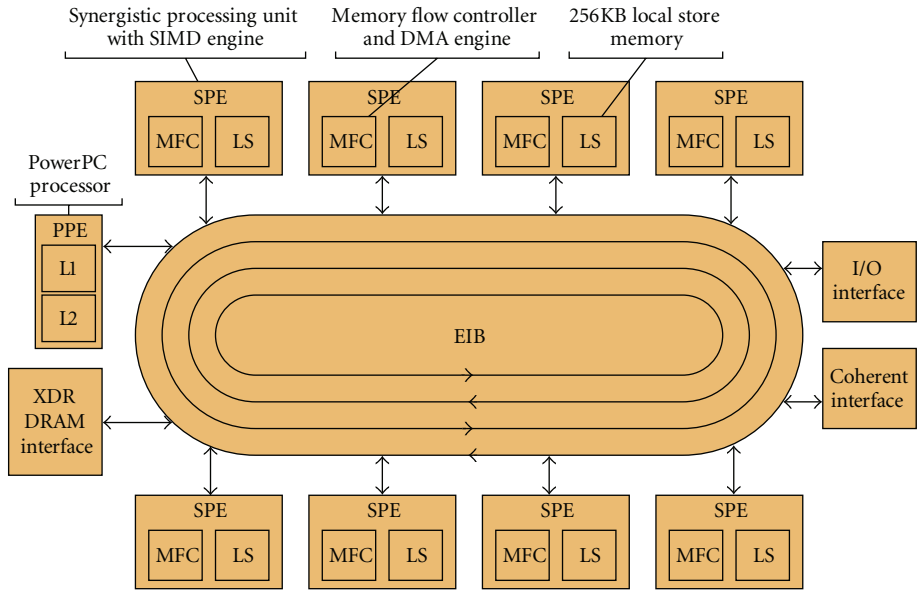



Figure 5: Architecture of Cell BE processor.

*2.4. The Cell BE Platform.* For the purpose of our Cell BE-based implementation, we used an IBM IntelliStation Z Pro Workstation with a Cell Acceleration Board (CAB). The CAB has one Cell Broadband Engine (Cell BE) running at 2.8 GHz, and 1 GB of XDR RAM. The IBM Cell BE is essentially a distributed memory, multiprocessing system on a single chip (see Figure 5). It consists of a ring bus that connects a single PowerPC Processing Element (PPE), eight Synergistic Processing Elements (SPEs), a high bandwidth memory interface to the external XDR main memory, and a coherent interface bus to connect multiple Cell processors together [13]. All these elements are connected with an on-chip Element Interconnect Bus (EIB). The first level instruction and data cache on the PPE are 32 KB and the level 2 cache is 512 KB. From a software perspective, the PPE can be thought of as the "host" or "control" core, where the operating system and general control functions for an application are executed.

The eight SPEs are the primary computing engines on the Cell processor. Each SPE contains a Synergistic Processing Unit (SPU), a memory flow controller, a memory management unit, a bus interface, and an atomic unit for synchronization mechanisms [14]. SPU instructions cannot access the main memory directly. Instead, they access a 256 KB local store (LS) memory, which holds both instructions and data. The programmer should keep all the codes and data within the size of LS and manage its contents by transferring data between off-chip memory and LS via mailboxes or direct memory access (DMA). This allows the programmer to overlap the computations and data transfer via double-buffering techniques [15].

We used Cell SDK version 3.0 and Mercury's MultiCore Framework (MCF) to develop our CellBE implementation. MCF uses a Function Offload Engine (FOE) model. In this model, the PPE acts as a manager directing the work of the SPEs. Sections of the algorithm in hand are loaded into the SPEs as individual "tasks." Data is then moved to the SPE where it is processed.

*2.5. The GPP Platform.* For the purpose of our GPP-based implementation of the Smith-Waterman algorithm, we targeted a PC with a 3.4 GHz Pentium 4 Prescott processor, 1 GB of RAM, running Windows XP OS. The Prescott processor has a 31 stage pipeline, 16 K 8-way associative L1 cache, and 1 MB L2 cache, and like all of the above platforms, it is also based on 90 nm CMOS technology.

## 3. Implementation of the Smith-Waterman Algorithm on FPGA

In this section, we will present the design of the Smith-Waterman algorithm implementation on FPGA. Figure 6 presents a linear systolic array for the implementation of a general purpose pairwise sequence alignment algorithm based on the dynamic programming algorithms presented in Section 2.1 above. The linear systolic array consists of a pipeline of basic processing elements ($PE_i$) each holding one query residue $x_i$, whereas the subject sequence is shifted systolically through the array [4]. Each PE performs one elementary calculation (see (3)) in one clock cycle and populates one column of the alignment matrix in turn (see Figure 7). The calculation at $PE_{i+1}$ depends on the result from $PE_i$, which means that each PE is one cycle behind its predecessor. The full alignment of two sequences of lengths $N$ and $M$ is hence achieved in $M + N - 1$ cycles.

The architecture of Figure 6 can cater for different sequence symbol types, sequence lengths, match scores, and matching task. Indeed, the sequence symbol type, for

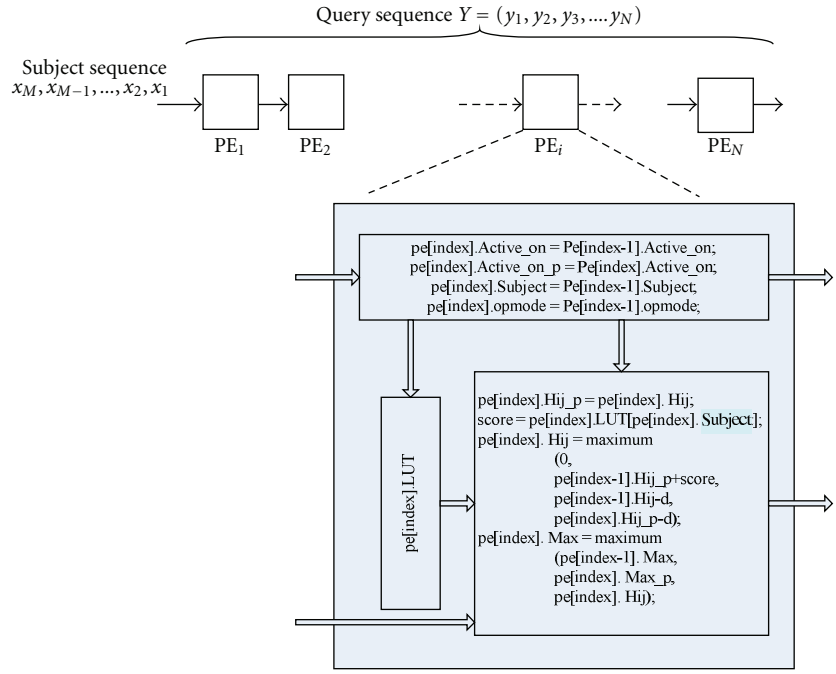



Figure 6: Linear processor array architecture for the FPGA implementation of the Smith-Waterman algorithm.

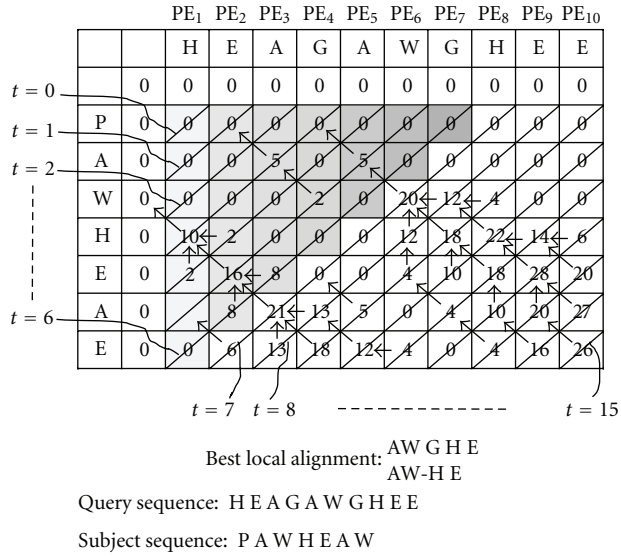



Figure 7: Illustration of the execution of the Smith-Waterman on the linear array processor.

example, DNA or Proteins, will only influence the word length of the input sequence, for example, 2 bits for DNA and 5 bits for Proteins, the query sequence length dictates the number of PEs, and the match score attributed to a symbol match depends on the substitution matrix used. Given a particular substitution matrix, for example, BLOSUM50, all possible match scores for a particular symbol represent one column in the substitution matrix. Hence, for each PE, we store the corresponding symbol's column in the substitution matrix, which we use as a look-up table. A different substitution matrix will hence simply mean a different look-up table content. The penalties attributed to a gap can also be stored in the PE.

The linear array of Figure 6 can also cater for different matching tasks with few changes. For instance, the difference between global alignment, local alignment, and overlapped matching [1, 4] resides in the initial values of the alignment matrix (border values), the recursive equation implemented by the PE, as well as the starting cell of the traceback procedure. Although a query sequence is often compared to a large set of database sequences, the traceback procedure is only needed for few sequences with high alignment scores. As such, it is customary to perform this on a host (sequential) processor as the time involved in this operation is negligible compared to the time it takes to align the query sequence against a whole sequence database.

*3.1. The Case of Long Sequences.* The number of PEs that could be implemented on an FPGA is limited by the logic resources available on-chip. For instance, the maximum number of PEs that could be implemented on a Xilinx XC2V6000 Virtex-II FPGA in the case of the Smith-Waterman algorithm with affine gap penalties is ~250. Clearly, this is not sufficient for many real world sequences where query sequence lengths can be in the thousands. The solution in such cases is to partition the algorithm in hand into small alignment steps and map the partitioned algorithm onto a fixed size linear systolic array (whose size is dictated by the FPGA in hand) as illustrated in Figure 8 below [4]. Here, the sequence alignment is performed in several passes. A First-In-First-Out (FIFO) memory block

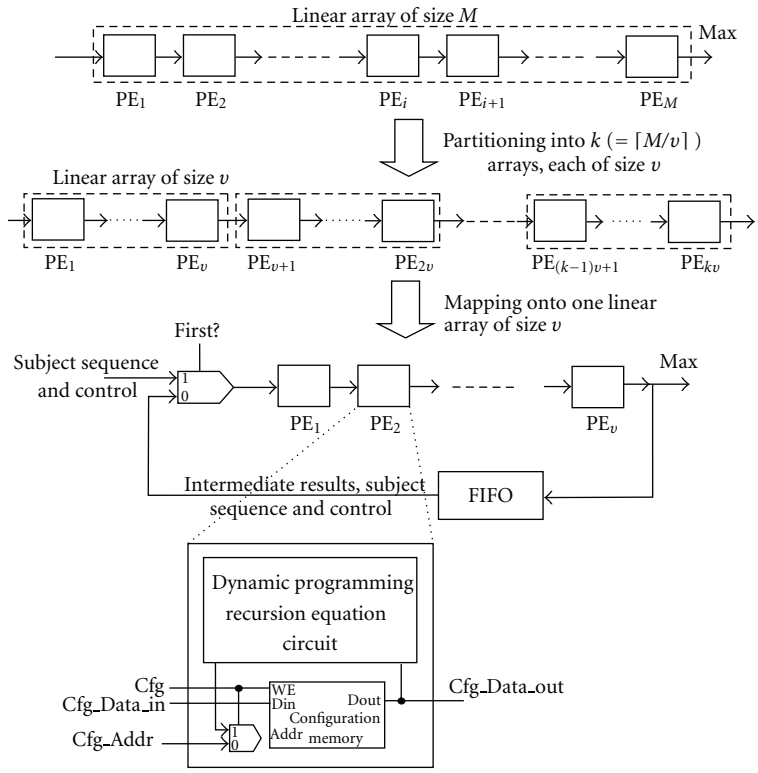



Figure 8: Partitioning/Mapping of a sequence alignment algorithm onto a fixed size systolic array.

is used to store intermediate results from each pass before they are fed back to the array input for the next pass. The depth of the FIFO is dictated by the length of the subject sequence. Another consequence of the folded architecture is that each PE should now hold $k$ substitution matrix columns (or look-up tables) instead of just one. In order to load the initial values of the look-up tables used by the PEs, a serial configuration chain is used, as illustrated in Figure 8. When the control bit *Cfg* is set to 1, the circuit is in configuration mode. Distributed memory in each PE then behaves as a register chain. Each PE configuration memory is loaded with the corresponding look-up tables sequentially. At the end of the configuration, *Cfg* is reset to 0 indicating the start of the operation mode.

## 4. Implementation of the Smith-Waterman Algorithm on GPU

The parallelization strategy adopted for our GPU implementation is based on multithreading and multiprocessing. Indeed, several threads are allocated to the computation of a single alignment matrix in parallel within a thread block, while several thread blocks are allocated to compute the alignments of different pairs of sequences [16]. We separate a single alignment matrix computation into multiple submatrices with a certain number of threads allocated to calculate each submatrix in parallel, depending on the maximum number of threads and maximum amount of memory available (see Figure 9). Once the allocated batch of threads completes a sub-matrix calculation, the final thread in the batch records the data in the row of which it takes charge and stores it into shared memory or global memory, depending on the size of database subject sequence, ready for the calculation of the next sub-matrix. The first thread in the batch, on the other hand, loads this data as initial data for the subsequent sub-matrix calculation. This operation continues in turn until the end of the entire alignment matrix calculation.

The above procedure makes this GPU implementation scalable to any sequence length. On the NVIDIA GeForce 8800GTX GPU, each SM can have 768 parallel threads running at the same time. Hence, we split this number into batches of threads or blocks, where each block computes one alignment matrix. For example, we can split the overall number of threads into 8 blocks of 96 threads, with 10 registers allocated to each thread and each block could use almost 2 KB of shared memory. Global memory will be used if this amount of allocated shared memory space is not enough for any database subject sequence. Note here that if the length of the database subject sequence is smaller than the number of threads in the block, additional waiting time should be added for the threads in the batch to finish their computations. This is easy to imagine, for example, if thread 0 has already completed its row calculation, but thread $n$ has not completed yet or has not even started its row, then thread 0 would have to wait for thread $n$ of the previous sub-matrix alignment to complete its task before obtaining its initial data for the sub-matrix alignment.

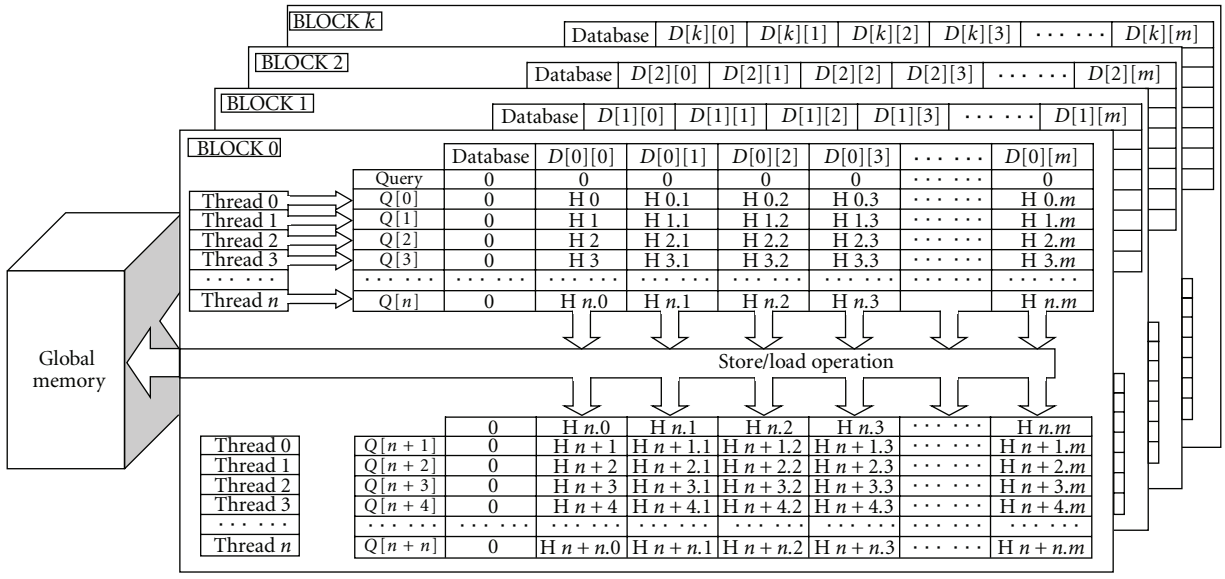



Figure 9: Our GPU parallel thread implementation of the Smith-Waterman algorithm: store and load operations are performed by the final thread and the first thread in each thread batch (block) to allow for any sequence length processing.

| | | | | | | |
|---|---|---|---|---|---|---|
| Thread 0 | $Q[0]$ | $D[0], D[1]$ | $D[2], D[3]$ | $D[4], D[5]$ | ⋯ ⋯ | $D[m-1], D[m]$ |
| Thread 1 | $Q[1]$ | $D[0], D[1]$ | $D[2], D[3]$ | $D[4], D[5]$ | ⋯ ⋯ | $D[m-1], D[m]$ |
| Thread 2 | $Q[2]$ | $D[0], D[1]$ | $D[2], D[3]$ | $D[4], D[5]$ | ⋯ ⋯ | $D[m-1], D[m]$ |
| ⋯ ⋯ | ⋯ ⋯ | ⋯ ⋯ | ⋯ ⋯ | ⋯ ⋯ | ⋯ ⋯ | $D[m-1], D[m]$ |
| Thread $n$ | $Q[n]$ | $D[0], D[1]$ | $D[2], D[3]$ | $D[4], D[5]$ | ⋯ ⋯ | $D[m-1], D[m]$ |

Figure 10: Alignment matrix calculation with vector variable char2.

*4.1. Load Partitioning Strategy.* In our GPU implementation, we used constant cache to store the commonly-used constant parameters in order to decrease access time, including the substitution matrix and the query sequence. In addition, we used global memory to store the database sequence as the size of the latter can be in the hundreds of Megabytes. Moreover, we used texture cache to shade database sequences. The bottleneck of speed-up in our implementation is the store operation of temporary data by the last thread and the load operation by the first thread in each batch, because the latency between SP registers and global memory is much longer than the one between registers and shared memory. No matter how fast other threads can execute the kernel code, they have to wait for a synchronization point of all threads. Obviously, this only occurs when the length of the database subject sequence is longer than the allocated space in shared memory. Therefore, our acceleration strategy mainly focuses on the efficient allocation of resources to each block to make the maximum use of the available parallelism. This can be achieved through setting the proper number of threads in each block. Here, since each SM has 8192 registers and can keep at most 768 parallel threads, for a query sequence of length 512, if we use 1 block of 512 threads, 16 registers can be used for each thread. In this case, only one pairwise sequence alignment can be computed in each SM. If we use 8 blocks of 64 threads, also 16 registers can be allocated to each thread, but the number of sequence alignments that can be processed at the same time becomes 8. Rather than adopting the simple method used in [17] which utilizes the full memory resources for each block, we flexibly allocate resources through setting the number of threads in each block, with no limitation on the overall length of the query sequence. Table 2 presents execution times of the Smith-Waterman algorithm on GPU using our technique, with different numbers of threads per block. For a query sequence of length 1023, 64 threads per block leads to the best performance.

Note finally that we used vector type *char2* as illustrated in Figure 10 to decrease the data fetch times compared to using *char* [12]. This was empirically found to be more efficient than using vector *char4*.

## 5. Implementation of the Smith-Waterman on IBM Cell BE

Our IBM Cell BE implementation exploits the data parallelism involved in aligning one query sequence with a large database of sequences by assigning different database

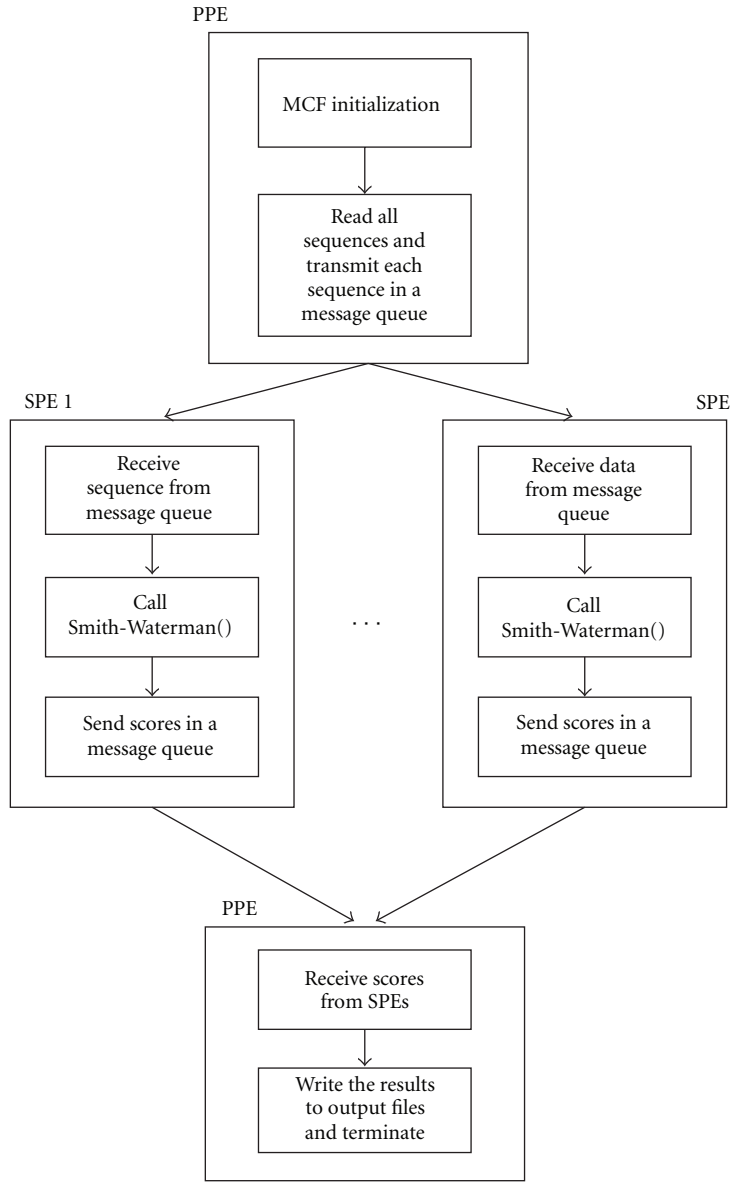



Figure 11: The parallel code flow of Smith-Waterman algorithm on Cell BE. The PPE transmits data to SPEs via one message queue and receives the results from SPEs by another.

sequences to the eight SPEs, that is, through multiprocessing. The PPE operates as a manager handling data I/O, assigning tasks and scheduling the SPEs. It reads input database sequences from disk, transmits different database sequences to different SPEs, and invokes SPEs to perform pairwise sequence alignment using the Smith-Waterman algorithm on independent sequences as illustrated in Figure 11.

The Cell BE uses two schemes to transfer data between the DRAM and the SPEs: (1) message queue and (2) direct memory access (DMA). In the message queue mode, the PPE reads the data from the DRAM, packetizes the data, and places packets into the message queue for the target SPE to read. DMA is a mechanism for an SPE to bypass the PPE and read data from the DRAM directly. This feature makes DMA a desirable option for data intensive applications. However, based on our timing trace results on the Cell BE, we found that the computation time within each SPE is the dominant component of the total execution time. For the case of query sequence length 256, we observed that 92.7% of the time is spent on computation and only 7.3% on the data transfer with the message queue. From this perspective, switching from message queue to DMA will not improve the performance considerably. Indeed, the average sequence length of the SWISS-PROT database is about 360, which can be completely packetized into a message queue and transmitted between PPE and SPEs. Therefore, we chose message queue as the parallelism strategy on the Cell BE due to the short bandwidth and latency of data communication [19].

Table 2: Performance comparison for different numbers of threads per block (64, 128, 256). All query sequences run against the SWISS-PROT database [18].

| Query length | Thread 64 time (sec) | Thread 128 time (sec) | Thread 256 time (sec) |
|---|---|---|---|
| 63 | 2.1 | 3.1 | 6.2 |
| 127 | 6.1 | 4.2 | 7.2 |
| 191 | 9.3 | 11.9 | 8.3 |
| 255 | 12.5 | 12.9 | 9.6 |
| 511 | 25.1 | 26.4 | 29.2 |
| 1023 | 50.4 | 53.1 | 57.8 |

We packetize and transmit each sequence in a message queue between PPE and SPEs. First, the manager and the workers all initialize the MCF network. Then, the PPE feeds the worker code into the local store memory of each SPE and signals them to start. As part of the initialization process, we dynamically set up two message queues, one is for PPE sending data to SPEs, and the other is backwards, for SPEs passing results back towards the PPE. After reading one sequence from the database, the manager puts it into one message queue and sets up a loop in which the PPE sends the message to SPEs separately. The manager then waits for the synchronization semaphore from the SPEs when they finish pulling the data into local store. Sequentially, the SPEs start processing the data in a concurrent manner. Whoever completes its computation first sends the results back to the PPE by means of the other message queue. This process continues until PPE transmits all the sequences to the SPEs. The manager then deallocates memory, destroys the MCF network, and terminates the program.

Parallelization of any algorithm requires careful attention to the partitioning of tasks and data across the system resources. Each SPE has a 256 KB local store memory that is allocated between executable code and data. The executable code section must contain the top-level worker code, the MCF functions, and any additional library functions that are used. If the total amount of executable code is too large for the allocated memory, it may be loaded as several "plug-ins." If the total amount of data exceeds the data allocation, it may be loaded down as "tiles." MCF contains plug-in and tile channel constructs to facilitate this as required. The tradeoff here is in increased code complexity. The core functions of the Smith-Waterman algorithm implementation compile to less than 83 KB. MCF adds up to 64 KB depending on the functions that are used. Rounding this up suggests that the worker code would somewhat be greater than half of the available SPE memory (128 KB). For our specific database, the maximum length of all the sequences is 35,213 bytes, which amounts to ~36 KB of data. These estimates suggest that each SPE could receive a full code segment and a complete set of protein sequence without the need for further partitioning.

Inside each SPE, a pairwise sequence alignment using the Smith-Waterman algorithm is performed column-wise, four cells at a time as illustrated in Figure 12 for a database sequence of length 4 and a query of length 8. When cells are calculated, we keep track of their updated values in a temporary register (cell calculations) which is updated each time a new column is calculated. The entire pairwise alignment matrix is not stored in memory, but rather just the temporary cell calculations column. Four dependency values are read at the beginning of an inner loop, and the new values for which the next column will be dependent are written at the end of the inner loop. The number of dependent cells needed for each alignment is simply equal to the length of the query sequence, since we are calculating cells column by column. After all SPE pairwise alignments are completed, the highest pairwise score calculated by each SPE is returned to the main program (in the PPE) for final reduction. The sequence with the highest score achieves the best alignment. Finally, it is worth mentioning that, currently, our query lengths are limited to 1024 residues, but we are working on some indexing strategies which will allow us to increase the length of a query.

## 6. Implementation of the Smith-Waterman Algorithm on GPP

Since our aim is to compare all four technologies not just in terms of speed, but also in terms of energy consumption, and purchase and development costs, we chose to use a widely adopted GPP implementation of the Smith-Waterman algorithm, namely, SSEARCH (version 35.04) from the FASTA set of programs [20]. SSEARCH was run on a 3.4 GHz Pentium 4 Prescott processor with 1 GB RAM, running Windows XP Professional. Using the SSEARCH program is perfectly justified for the purpose of this paper since it is a mature piece of software that is widely used by Bioinformaticians in practice. We are aware of better GPP implementations in the literature, for example, [21]. However, here again, such implementations do not give us the development time, for instance, nor do they guarantee a fair balance of experience between the developers of each implementation. They hence do not serve the particular aims of this paper.

## 7. Comparative Implementation Results and Evaluation

This section presents the implementation results of our Smith-Waterman designs on FPGA, GPU, Cell BE, and GPP. Table 3 first presents the execution times of the Smith-Waterman implementation on all four platforms for a number of query sequences against the SWISS-PROT database (as of August 2008) when it contained 392,768 sequences and a total of 141,218,456 characters. For the FPGA, GPU, and Cell BE implementations, we assume that the database is already on the accelerator card's memory. Thus, the execution times shown in Table 3 do not include the database transfer time as it is an initial step. In practice, queries are made against fairly static databases, and hence this assumption is reasonable.

Note that for smaller sequences, the target FPGA chip could easily fit more processing elements on chip and thus

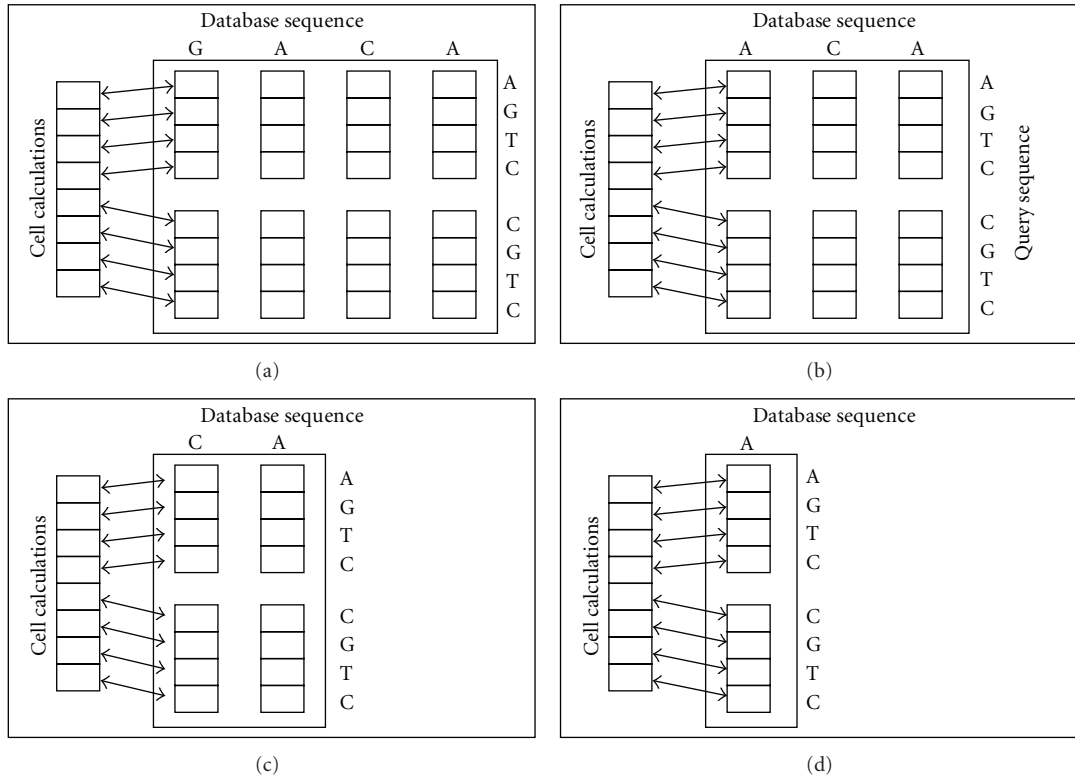



Figure 12: Illustration of the Smith-Waterman calculation on the Cell BE.

Table 3: Performance comparison. All query sequences run against the SWISS-PROT database.

| Query (protein name) | Query length | FPGA time (sec) | GPU time (sec) | Cell BE time (sec) | GPP time (sec) |
|---|---|---|---|---|---|
| P36515 | 4 | 1.5 | 4.1 | 0.5 | 24 |
| P81780 | 8 | 1.6 | 4.1 | 1.0 | 30 |
| P83511 | 16 | 1.6 | 4.3 | 1.3 | 43 |
| O19927 | 32 | 1.6 | 4.7 | 1.4 | 62 |
| A4T9V0 | 64 | 1.6 | 6.7 | 2.5 | 115 |
| Q2IJ63 | 128 | 1.6 | 12.8 | 5.1 | 210 |
| P28484 | 256 | 1.9 | 30.0 | 9.4 | 424 |
| Q1JLB7 | 512 | 4.5 | 76 | 17.2 | 779 |
| A2Q8L1 | 768 | 6.7 | 136.2 | 22.2 | 1356 |
| P08715 | 1024 | 8.9 | 172.8 | 31.8 | 1817 |

(provided there is enough bandwidth to transfer more data to the FPGA chip in parallel) the execution time could be reduced several fold. A fairer comparison in speed would take the results of sequence length 256 since it is close to the average sequence length in the SWISS-PROT database (360). Each PE in the FPGA systolic array consumes ~110 slices, and, consequently, we were able to fit ~500 PEs on a Xilinx Virtex-4 LX160-11 FPGA [22]. Moreover, the processing word length in the FPGA systolic array is 16 bits, and the circuit was clocked at 80 MHz. Table 4 presents the corresponding performance figures in Giga Cell Updates Per Seconds (GCUPS) (the CUPS (or Cell Updates Per Second) is a common performance measure used in computational biology. Its inverse represents the equivalent time needed for a complete computation of one entry of the alignment matrix) as well as the speed-up figures normalized with respect to the GPP implementation result. This shows the FPGA solution to be two orders of magnitude quicker than the GPP solution, with the Cell BE and GPU coming second and third, respectively. The latter two achieve one order of magnitude speed-up compared to GPP.

We note before embarking on result evaluation that faster implementations do exist in the literature. For instance, in [21], the author presented a GPP implementation of the Smith-Waterman algorithm on a 2.0 GHz Xeon Core 2 Duo processor with 2 GB of RAM running Windows

Table 4: Performance comparison for query sequence of length 256.

| Platform | GCUPS | Speed-up |
|---|---|---|
| FPGA | 19.4 | 228 : 1 |
| GPU | 1.2 | 14 : 1 |
| Cell BE | 3.84 | 45 : 1 |
| GPP | 0.085 | 1 : 1 |

Table 5: Development times of the Smith-Waterman algorithm implementation on all four technologies.

| Platform | Development time in days |
|---|---|
| FPGA | 300 |
| GPU | 45 |
| Cell BE | 90 |
| GPP | 1 |

XP SP2. The software implementation exploited Intel SSE2 technology and resulted in a much higher performance of 1.37 GCUPS. Moreover, a Smith-Waterman implementation on an NVIDIA GTX 295 Dual Core GPU, which contains 30 SPs, 896 MB memory per GPU core, installed on a PC with a Core 2 Duo E7200 2.53 GHz processor, with 2 GB RAM, and running Cent OS 5.0, resulted in ∼11 GCUPS performance [23]. This shows that GPPs and GPUs can outperform the above results considerably with more design effort, for example, exploiting SSE2 technology in [21], and optimizing thread scheduling and memory architecture [23] as well as exploiting more advanced process technologies (below 90 nm). However, the aim of this paper is to perform a fair comparison of all four technologies in terms of speed, development time, cost, and energy consumption. For instance, the process technology of the GPP and GPU devices reported in [21, 23], respectively were more advanced than the FPGA technology we used in this study. Moreover, these implementations do not report the development time which is crucial to assess productivity as will be shown below. As such, the following will concentrate on the results shown in Table 4 above rather than other implementations reported in the literature as these do not serve the particular aims of this paper, despite their worth.

In order to put the speed-up figures shown in Table 4 into perspective, we measured the time it took to develop each of these implementations. Indeed, each of the four implementations was developed by a different Ph.D. student with a comparable experience in programming his/her respective platform. Table 5 presents the resulting development times.

This shows FPGA development time to be one order of magnitude higher than that of Cell BE and GPU, and two orders of magnitude higher than that of GPP. It is worth mentioning that the majority of FPGA development time (∼80%) was spent in learning the specific FPGA hardware application programming interface (API) as well as debugging the FPGA implementation in hardware. As such the choice of the hardware description language (e.g., VHDL or Verilog instead of Handel-C) in itself would not have had a major impact on the figures. The lack of standards (e.g., standard FPGA hardware boards, standard FPGA APIs) however remains a major problem for FPGA programmers.

Table 6: Cost of purchase and development of the Smith-Waterman algorithm implementation on all four technologies.

| Platform | Purchase cost ($) | Development cost ($) | Overall cost ($) | Normalized overall cost |
|---|---|---|---|---|
| FPGA | 10,000 | 48,000 | 58,000 | 50 |
| GPU | 1450 | 7,200 | 8,650 | 8 |
| Cell BE | 8,000 | 14,400 | 22,400 | 19 |
| GPP | 1000 | 160 | 1160 | 1 |

Table 7: Performance per $ spent for each technology.

| Platform | Performance (MCUPS) per $ spent | Normalized performance per $ spent |
|---|---|---|
| FPGA | 0.34 | 4.6 |
| GPU | 0.14 | 1.9 |
| Cell BE | 0.17 | 2.3 |
| GPP | 0.07 | 1 |

By accounting for the cost of development (measured on the basis of US$20/hour as the average salary of a freshly graduated student where the experiments took place) and the cost of purchase of the respective platforms, Table 6 gives the overall development cost of all four solutions. Note here that the purchase cost of FPGA, GPU and Cell BE includes the cost of the host machine.

We can see that the FPGA solution is 50x more expensive than the GPP solution, followed by the Cell BE (19x) and the GPU (8x). Based on these figures, we can measure the performance per dollar spent by dividing the GCUPS figures of Table 4 by the overall cost figures given in Table 6 for each platform. The results are presented in Table 7 below (performance is expressed in Mega CUPS per dollar).

This shows the FPGA platform to be a more economical solution for this particular algorithm despite its relatively high cost, thanks to its much higher performance. The CellBE and GPU came second and third, respectively.

We have also measured the power consumed by each implementation (excluding the host in the case of FPGA, GPU, and Cell BE) as shown in Table 8. We used a power meter connected between the power socket and the machine under test for this purpose. We noted the power meter reading, at steady state, when the Smith-Waterman algorithm was running. This includes two parts: an idle power component and a dynamic power component. The idle power component can be obtained from the power meter when the machine is in the idle state, which means that no Smith-Waterman algorithm implementation was running on it. The dynamic power consumption is thus obtained by deducting the idle power reading from the steady state power reading. The power measurement results are shown in Table 8.

Table 8: Power consumption of the Smith-Waterman algorithm implementation on all four technologies.

| Platform | Idle power (watt) | Steady state power (watt) |
|---|---|---|
| FPGA (clocked at 80 MHz) | 100 | 139 |
| GPU | 70 | 126 |
| Cell BE | 180 | 140 |
| GPP | 70 | 100 |

Table 9: Power and energy consumption of the Smith-Waterman algorithm implementation on all four technologies.

| Platform | Power (watt) | Energy (joule) | Normalized energy consumption |
|---|---|---|---|
| FPGA (clocked at 80 MHz) | 39 | 73 | 0.0017 |
| GPU | 56 | 1682 | 0.04 |
| Cell BE | 140 | 1317 | 0.03 |
| GPP | 100 | 42400 | 1 |

Table 10: Performance per watt figures of the Smith-Waterman algorithm implementation on all four technologies.

| Platform | Performance (MCUPS) per watt | Normalized performance per watt |
|---|---|---|
| FPGA | 508 | 584 |
| GPU | 22 | 25 |
| Cell BE | 27 | 31 |
| GPP | 0.87 | 1 |

We use the dynamic power figures for the accelerated implementations, that is, the FPGA, GPU, and Cell BE-based implementations, as nearly all of the processing is done on the accelerator, with the host only sending query data and collecting results from the accelerator. As such, the cost and power consumption of the host could be made as small as needed without affecting the overall solution performance. The GPP implementation's steady state power figure however is used, instead of the dynamic power, as there is no distinction between host and accelerator in this case.

Multiplying the power figure for each platform with the execution time, we obtain the energy consumed by each implementation as shown in the Table 9.

This shows the FPGA solution to be three orders of magnitude more energy efficient than GPP, while the Cell BE and the GPU came second and third, respectively (with one order of magnitude energy efficiency compared to GPP). The performance per watt figure can thus be calculated by dividing the GCUPS figures of Table 4 by the power consumption figure in Table 9 for each platform. The results are presented in Table 10 (performance is expressed in Mega CUPS per watt).

This again highlights the high energy efficiency of the FPGA solution, followed by Cell BE and GPU. The latter is often unfairly characterized as energy inefficient in the computing community, something that the results of this study dispute. Indeed, factoring the speed-up gains, GPUs can be much more energy efficient than GPPs, as shown in this study.

Table 11: Performance per $ and per watt for each technology using the GPP implementation of [21] and GPU implementation of [23].

| Platform | Performance (MCUPS) per $ | Performance (MCUPS) per watt |
|---|---|---|
| FPGA | 0.34 | 508 |
| GPU | 1.27 | 196 |
| Cell BE | 0.17 | 27 |
| GPP | 1.18 | 13.7 |

It is important to note at this stage that the above results are very sensitive to the technology used and level of effort spent on the implementation. For instance, if we consider the GPP and GPU implementations reported in [21, 23], respectively, and assuming that development times and power consumption figures were similar to the GPP and GPU implementations reported in this paper, then the resulting performance per $ and performance per watt figures of the GPP and GPU implementations would have been as shown in Table 11.

This shows GPUs to be more economic on performance per $ grounds compared with other technologies, followed closely by GPPs. Such conclusion however is not valid according to the criteria set in this paper, since the GPP and GPU implementations reported in [21, 23], respectively were based on more advanced process technologies, brought to market after Virtex-4 FPGAs. In addition, these implementations needed more design effort. Nonetheless, we note that the performance per watt figures in Table 11 still show FPGAs to be far superior to the other technologies on energy efficiency criterion.

We note finally that the Smith-Waterman algorithm implementation scales extremely well with data sizes and computing resources with the four technologies used (FPGAs, GPUs, Cell BE, or GPP). Indeed, the algorithm is characterized by high level data and instruction level parallelism, and given the parameterizable way in which we designed our solutions, the same piece of code can be used to take advantage of increasing resources on the target platform, for example, FPGAs with more slices and memory, GPUs with more stream processors and threads, Cell BEs with more SPEs. In the case of GPUs, for instance, allocating different pairwise alignments to extra GPU stream processors would increase the speed-up proportionally, assuming memory bandwidth is also increased. Beyond a single chip, a straightforward way of scaling up the algorithm is to split the subject database into $N$ subdatabases and allocate each subdatabase to one GPU chip. Partial results are then reduced by a host processor in a typical scatter-gather manner, as demonstrated in [23]. The same reasoning applies to general purpose processors and Cell BE. As for FPGAs, bigger chips would result in an increase of the number of PEs that could fit on chip, which would in turn increase

the GCUPS performance proportionally, provided proper control circuitry is employed to use all PEs (or a large proportion of them) at any given time. Such techniques were illustrated in [5]. In general, and given the high computation to communication ratio of the algorithm, the scalability of the execution time as a function of available hardware resources is near linear.

In view of the above, the following conclusion section summarizes the findings of this study and presents a number of general lessons learnt through it.

## 8. Conclusion

This paper showed the design and parallel implementation of the Smith-Waterman algorithm on three different technologies: FPGA, GPU, and IBM Cell BE and compared the results with a standard sequential GPP implementation. Comparison criteria included speed, development time, overall cost, and energy consumption. This study revealed FPGAs to be a cost effective and energy efficient platform for the implementation of the Smith-Waterman algorithm as it came on top on both performance per dollar and performance per watt criteria. FPGAs achieved 4.6x more performance for each $ spent and 584x more performance for each Watt consumed, compared to GPPs. The IBM Cell BE came second as it achieved 2.3x more performance for each $ spent and 31x more performance for each Watt consumed, compared to CPUs. Finally, the GPU came third as it achieved 1.9x more performance for each $ spent and 25x more performance for each watt consumed, compared to GPPs.

The speed of FPGA implementation was limited by the amount of logic resources on chip as more parallelism could be obtained with more processing elements. As for the GPU, the parallelism was limited by the size of local memory (shared memory and number of registers) as well as the number of parallel processes and threads that could be launched at the same time, which is dictated by the number of stream processors and their parallelism potential. Finally, for the Cell-BE, the parallelism was mainly limited by the number of Synergistic Processing Elements (SPEs) and the parallelism potential of each SPE. Specifically, with the Cell BE, it is not feasible to match the fine grained parallelism level of the GPU. Indeed in the Cell BE implementation, we divided the workload equally among the SPEs and let each SPE run the sequence alignment algorithm on its own data set. If an SPE is assigned to process "$n$" sequences, the program is executed over these one after the other, in a sequential manner.

We note, however, that the overall cost of the implementations did not include the energy cost as this would depend on the amount of use of the platform. More importantly perhaps these calculations did not account for issues such as technology maturity, backward and forward compatibility, and algorithms' rate of change, which all play an important role in technology procurement decisions. Unfortunately, these issues are more difficult to quantify and are often subjective and time/location-sensitive. We also note that the comparison presented in this paper has been conducted for one single algorithm, which limits the generalizability of the results. Indeed, we recognize that our experiment is not statistically significant and that development times for instance would vary significantly for one programmer to another. Nonetheless, our objective was to put the "speed-up" values achieved into perspective with an analysis of productivity based on the personal experience of an average Ph.D. student. As such, the following general lessons could be learnt from the case study presented in this paper.

First, the reason FPGAs outperform other architectures in this Smith-Waterman case study is three-fold: (1) the high level of data and instruction level parallelism available in the algorithm which suits FPGA distributed resources very well, (2) the fine granularity of the instructions involved which suits the fine-grained FPGA computing resources (e.g., abundant 4-bit lookup tables and shift registers), and (3) the relatively low dynamic range requirement, which means that fixed point arithmetic with a relatively small number of bits can be used, which also suits FPGAs' fine-grained architectures. Thus, any application with high fine-grained instruction and data level parallelism, and modest dynamic range data requirements should be expected to achieve similar performance gains on FPGAs. General purpose processors should be expected to fare better for less parallel/regular algorithms on the other hand. Second, FPGA technology's main competitive advantage is on performance per watt criteria. High performance computing applications where power consumption is often a bottleneck should hence benefit from this technology. Third, on the economic viability front (i.e., performance per dollar spent), and using currently available hardware and development tools, FPGAs need to achieve at least two orders of magnitude speed-up compared to GPPs and one order of magnitude speed-up compared to GPUs and CellBE to justify their relatively longer development times and higher purchase costs. This relatively high hurdle represents a major problem in the face of further market penetration for FPGA technology, and it is mainly due to FPGAs' relatively longer development times. Standard FPGA boards with standard communication and application programming interfaces can lower the aforementioned hurdle drastically as the majority of FPGA development time is often spent on learning and debugging specific FPGA hardware application programming interfaces and tools.

## References

[1] R. Durbin, S. Eddy, S. Krogh, and G. Michison, *Biological Sequence Analysis: Probabilistic Models for Proteins and Nucleic Acids*, Cambridge University Press, 1998.

[2] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[3] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[4] K. Benkrid, Y. Liu, and A. Benkrid, "A highly parameterized and efficient FPGA-Based skeleton for pairwise biological

sequence alignment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 4, pp. 561–570, 2009.

[5] T. Oliver, B. Schmidt, and D. Maskell, "Hyper customized processors for bio-sequence database scanning on FPGAs," in *Proceedings of the ACM/SIGDA 13th ACM International Symposium on Field Programmable Gate Arrays (FPGA '05)*, pp. 229–237, February 2005.

[6] G. A. Harrison, J. M. Tanner, D. R. Pilbeam, and P. T. Baker, *Human Biology: An Introduction to Human Evolution, Variation, Growth, and Adaptability*, Oxford Science, 1988.

[7] HP Proliant DL145 Server Series, Hewlett-Packard, 2007, http://www.hp.com/.

[8] Celoxica, The RCHTX FPGA Acceleration Card Data Sheets, Celoxica, 2007, http://www.hypertransport.org/docs/tech/rchtx_datasheet_screen.pdf.

[9] Xilinx Corporation, Virtex-4 Family Data Sheets, 2007, http://www.xilinx.com/support/documentation/virtex-4_data_sheets.htm.

[10] Agility DS, Handel-C Reference Manual, 2009, http://www.mentor.com/products/fpga/handel-c/upload/handelc-reference.pdf.

[11] Nvidia Corporation, GeForce 8800 GPUs, 2009, http://www.nvidia.co.uk/page/geforce_8800.html.

[12] CUDA, CUDA Programming Guide Version 1.1, NVIDIA Corporation, 2009, http://developer.nvidia.com/cuda/.

[13] Mercury Computer Systems, Datasheet: Cell Accelerator Board, 2009, http://www.mc.com/.

[14] F. Blagojevic, D. S. Nikolopous, A. Stamatkis, and C. D. Antonopoulos, "Dynamic multigrain parallelization on the cell broadband engine," in *Proceedings of the ACM SIGPLAN Principles and Practice of Parallel Computing (PPoPP '07)*, San Jose, Calif, USA, March 2007.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 589–604, 2005.

[16] C. Ling, K. Benkrid, and T. Hamada, "A parameterisable and scalable smith-Waterman algorithm implementation on CUDA-compatible GPUs," in *Proceedings of the IEEE 7th Symposium on Application Specific Processors (SASP '09)*, pp. 94–100, July 2009.

[17] Y. Munekawa, F. Ino, and K. Hagihara, Design and Implementation of the Smith-Waterman Algorithm on the CUDA-Compatible GPU, 2008.

[18] A. Bairoch and R. Apweiler, The SWISS-PROT protein knowledgebase and its supplement TrEMBL, *Nucleic Acid Research, Release* 56.3, October 2008.

[19] Y. Song, G. M. Striemer, and A. Akoglu, "Performance analysis of IBM Cell Broadband Engine on sequence alignment," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '09)*, pp. 439–446, August 2009.

[20] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 85, no. 8, pp. 2444–2448, 1988.

[21] M. Farrar, "Striped Smith-Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[22] Y. Liu, K. Benkrid, A. Benkrid, and S. Kasap, "An FPGA-based web server for high performance biological sequence alignment," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '09)*, pp. 361–368, August 2009.

[23] K. Dohi, K. Benkrid, C. Ling, T. Hamada, and Y. Shibata, "Highly efficient mapping of the Smith-Waterman algorithm on CUDA-compatible GPUs," in *Proceedings of the 21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP '10)*, pp. 29–36, July 2010.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 564704, 15 pages
doi:10.1155/2012/564704

*Research Article*

# An Evaluation of an Integrated On-Chip/Off-Chip Network for High-Performance Reconfigurable Computing

**Andrew G. Schmidt,[1] William V. Kritikos,[2] Shanyuan Gao,[2] and Ron Sass[2]**

[1] *Information Sciences Institute, University of Southern California, 3811 North Fairfax Drive, Suite 200, Arlington, VA 22203, USA*
[2] *Reconfigurable Computing Systems Lab, UNC Charlotte, Electrical and Computer Engineering Department, 9201 University City Boulevard, Charlotte, NC 28223, USA*

Correspondence should be addressed to Andrew G. Schmidt, andrewgschmidt@gmail.com

Received 28 September 2011; Accepted 18 February 2012

Academic Editor: Esam El-Araby



As the number of cores per discrete integrated circuit (IC) device grows, the importance of the network on chip (NoC) increases. However, the body of research in this area has focused on discrete IC devices alone which may or may not serve the high-performance computing community which needs to assemble many of these devices into very large scale, parallel computing machines. This paper describes an integrated on-chip/off-chip network that has been implemented on an all-FPGA computing cluster. The system supports MPI-style point-to-point messages, collectives, and other novel communication. Results include the resource utilization and performance (in latency and bandwidth).

## 1. Introduction

In 2007 the *Spirit* cluster was constructed. It consists of 64 FPGAs (no discrete microprocessors) connected in a 3D torus. Although the first integrated on-chip/off-chip network for this machine was presented in 2009 [1], the design has evolved significantly. Adjustments to the router and shifts to standard interfaces appeared as additional applications were developed. This paper describes the current implementation and the experience leading up to the present design. Since the network has been implemented in the FPGA's programmable logic, all of the data presented has been directly measured; that is, this is not a simulation nor emulation of an integrated on-chip/off-chip network.

A fundamental question when this project began was whether the network performance would continue to scale as the number of nodes increased. In particular, there were three concerns. First, would the relatively slow embedded processor cores limit the effective transmission speed of individual links? Second, were there enough resources? (Other research has focused on mesh-connected networks rather than crossbars due to limited resources [2–5].) Third, would the on-chip and off-chip network bandwidths be balanced so one does not limit the other? Although some of the data presented here has appeared in publications related to different aspects of the project, the aim of this paper is to provide a comprehensive evaluation of the on-chip/off-chip network. The results are overwhelmingly positive, supporting the hypothesis that the current design is scalable.

The rest of this paper is organized as follows. In the next section some related work is presented for on-chip networks. In Section 3 we describe the reconfigurable computing cluster project and the small-scale cluster, *Spirit*. Following that, in Section 4, the specifics of the present on-chip/off-chip network design are detailed. The next three sections describe the performance of the network under different scenarios. Specifically, Section 5 shows how MPI collective communication cores implemented in hardware can be interfaced directly to the network to improve the performance of message-passing applications in software. Section 6 demonstrates application-specific hardware cores interfacing to the network. In Section 7, a novel performance monitoring system is presented that uses the network with

minimal invasion to the processor. The paper concludes with Section 8 where a summary of the results thus far and future work are described.

## 2. Related Work

The idea of using a network on chip to replace global wiring was proposed by Dally and Towles [6] in 2001. That paper argued that the amount of resources required to implement the network (they estimated 6.6% of the chip) was small relative to the benefits which include opportunities for optimization and a universal interface (to promote reuse). Since then, most of the research in this area has taken a VLSI focus [7], looking at issues such as power, speed, and flow control [8, 9].

Nearly all of the academic research has focused on a 2D mesh or torus topology. This is a natural extension of the medium—routers are easily connected by wires allocated in the upper two metal layers. However, in industry, the network of choice is currently the complete graph ($K_n$ where $n$ is the number of cores); that is, all cores are directly connected to all other cores. Intel's Quick Path Interconnect exemplifies this approach. Industry acknowledges that this is an interim solution, but the public roadmap does not state what will replace it.

Although the body of academic research generally does not discuss interfacing the on-chip network to an external network, one can anticipate the current path. Increasingly, chip manufacturers are incorporating new cores, such as graphic processors, memory controllers, and network interfaces. VLSI researchers generally see the NoC as connecting modules of the chip. The natural eventuality would be to connect the network interface to the NoC as just another module. What sets the network presented here apart from this approach are two factors. (1) Unrelated traffic can cause contention that blocks messages destined for the network interface leading to jitter. (2) Multiple messages destined for off-chip locations have to be serialized. In summary, these related works rely on simulations with built-in assumptions about general-purpose traffic patterns, but HPC traffic has a bias towards the network interface.

## 3. Reconfigurable Computing Cluster

The reconfigurable computing cluster (RCC) project [10] is investigating the role (if any) field programmable gate arrays (FPGAs) have in the next generation of very-large-scale parallel computing systems. Faced with growing needs of computational science and the existing technology trends, the fundamental question is *how to build cost-effective high-performance computers?* To answer this question, the RCC project identified four research areas to investigate (which recently have gained attention [11]), namely, memory bandwidth, on-chip and off-chip networking, programmability, and power consumption. While the focus here is on the networking, the three remaining categories clearly play a significant role and, as a result, will also be present in the discussion.

There are numerous aspects to FPGAs that make them especially useful for high-performance computing. One advantage is that, because a single FPGA is now large enough (and has a rich enough set of on-chip IP), it is able to host an entire system on chip. This eliminates many of the peripheral components found on a standard commodity PC board which in turn offers size, weight, and power advantages. Also, by reducing the size of the node, there is a secondary advantage in that it reduces the distance between nodes enabling novel networking options. Research also indicates that FPGA floating-point performance will continue to rise (both in absolute terms and relative to single-core processors [12]).

Certainly, there are potential pitfalls that could limit the expected performance gains of such an approach. Until recently, FPGAs have had the reputation of being slow, power hungry, and not very good for floating-point applications [13]. However, these generalizations are based on a snap shot in time and depend heavily on the context in which the FPGA is employed. Moreover, while a number of HPC projects are using (have used) FPGAs [14–22], they have been used as "compute accelerators" for standard microprocessors. The RCC project's proposed approach is fundamentally different in that the FPGAs are promoted to first class computation units that operate as peers. The central hypothesis of the project is that the proposed approach will lead to a more cost-effective HPC system than commodity clusters in the high-end computing range.

The RCC's initial investigation began in early 2007 with the assembly of *Spirit*, a small-scale cluster of 64 all-FPGA compute nodes (with no discrete microprocessors in the design), arranged in a 4-ary 3-cube network topology. Each compute node consists of a Xilinx ML410 development board [23] with a Virtex-4 FX60 FPGA, 512 MB of DDR2 memory, gigabit ethernet this time only and a custom high speed network (among several other peripherals). Figure 1 depicts a simple representation of the organization of the cluster. The server node is a commodity x86 server used to manage the cluster and provide a network filesystem for the cluster. The server node connects to the cluster through a commodity Gigabit Ethernet switch to which each FPGA node also connect.

Each node receives an application-specific configuration bitstream over the Ethernet network through a custom application called FPGA session control (FSC) [24]. (This network is just used for administrative traffic, such as rsh/ssh, ntp, NFS, etc.) FSC enables collaborates to access the Spirit cluster remotely and upload ACE files to each node's CompactFlash. The user can then select a specific ACE file to boot from, which in turn configures the FPGA with the intended design. Further details regarding the network will be discussed in Section 4.

## 4. Spirit's Integrated On-Chip and Off-Chip Network

At the heart of the spirit cluster is the architecture independent reconfigurable network (AIREN) which is

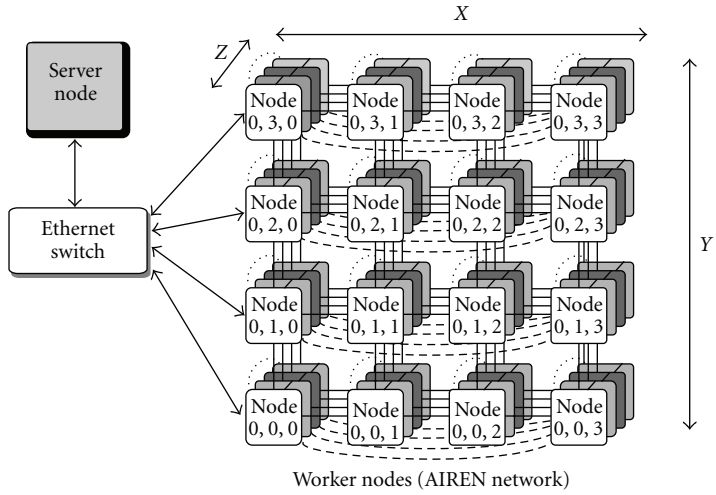



Figure 1: The RCC's Spirit cluster consists of 64 FPGAs connected in a 4-ary 3-cube through a high-speed network (AIREN).

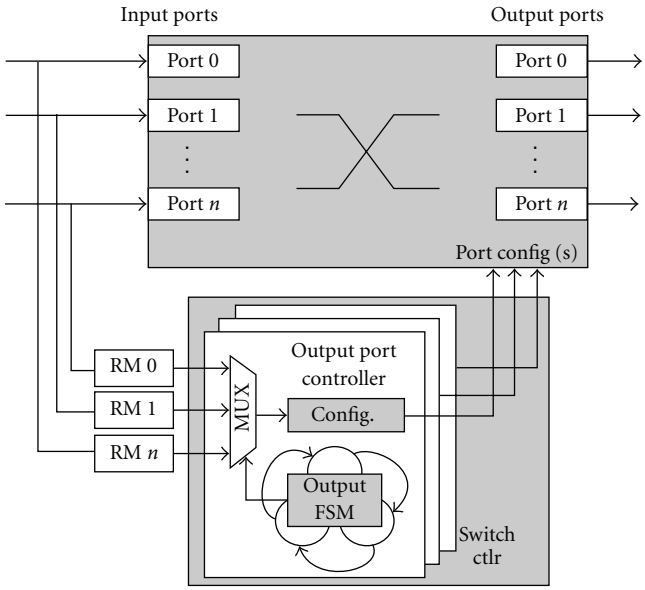



Figure 2: Block diagram of the AIREN router.

an integrated on-chip/off-chip network that not only supports efficient core-to-core communication but node-to-node as well. Figure 14 shows a high-level overview of the FPGA's system on chip. On the FPGA, AIREN spans several IP cores, of which the AIREN router is the most central component. The original implementation of AIREN has been published in a short paper in 2009 [1]; however, significant architecture improvements have been made which warrant further discussion.

AIREN offers several configurability options in order to provide the desired network behavior. These include the ability to change the number of input and output ports (radix) of the switch, the data width, operating frequency, whether the switch has software controlled routing or hardware accelerated routing, the dimensionality of the off-chip network (ring, mesh, cube, etc.), and the routing methodology. These details and more regarding the AIREN network will be provided within this section.

*4.1. AIREN Router.* Figure 2 details the internal components and functionality of the AIREN router. The router consists of a single crossbar switch implemented in the FPGA's programmable logic along with routing decision modules that inspects the header of each packet and passes the routing decision to the switch controller. The switch controller manages the connections and deals with contention for output ports. The crossbar switch is implemented as a generic VHDL model to allow easy customization of the number of ports needed for a given implementation. The signals that are switched include 32 bits of data along with 4 bits of flow control signals. On-chip cores connect to the switch for a high-bandwidth, low-latency communication. Off-chip cores communicate through a set of special ports which interface to the FPGAs multi-gigabit transceivers for communication with other FPGAs.

*4.2. AIREN Interface.* One of the first architectural enhancements to the AIREN router is changing from the use of an in-house interface (AIREN network interface) to connect compute cores to the network. A consequence of that interface was less than optimal latency of 0.08 $\mu$s across the switch. More recently, work has been done to migrate to a more widely used and accepted interface standard supported by Xilinx known as LocalLink [25]. With the adoption of this standard, compute cores can be quickly connected to the network. In addition, LocalLink provides flow control which reduces the need to incorporate buffers throughout the network, freeing up scarce on-chip memory resources (BRAM). Furthermore, LocalLink supports frames, which means headers and footers are more easily identifiable and results in a reduction of the latency across the switch to 0.02 $\mu$s. Figure 3 provides a simple diagram of how the

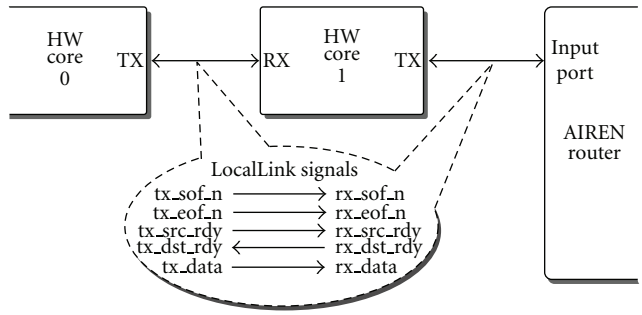



Figure 3: Simple illustration to show the Xilinx LocalLink standard implemented within the AIREN network to support both core-to-core and core-to-router connectivity.

LocalLink standard is incorporated into the network and specifically can allow compute cores to be chained together and/or connected to the AIREN router. LocalLink is used in both the software- and hardware-controlled routing configurations.

*4.3. AIREN Routing Module.* Each input port connects to both a port on the crossbar switch and to a routing module hardware core. The routing module examines the header of each incoming packet to determine its destination. Each node receives a unique node ID, and each core has a core id that is only required to be unique on each node, a decision that was made to simplify routing to first route to a node and then route to a core. The routing module looks at the packet header and identifies the packet's destination in terms of `node ID` and `core ID`. If the packet's node ID matches the local node ID, then the core ID is used to route the packet to the corresponding core. If the IDs differ, the routing module directs the packet to the corresponding node's off-chip port, depending on the routing algorithm. In the work presented here, a simple dimension-ordered routing algorithm is used to determine which outgoing port the message is routed to [6], although the routing module is configurable to support different off-chip routing algorithms. Once a routing decision has been made, the routing module passes the routing request to the switch controller and must wait until it is granted access to the crossbar switch. When an input port is granted access to the output port, it owns that port until the packet is finished.

A packet includes within the first 8-byte header both the node and core ID, as can be seen in Figure 4. The LocalLink start-of-frame (SOF) signal marks the beginning of a packet. The routing module analyzes the header to make the routing decision; otherwise the routing module is a passive component in the system. At present no footer is used; however, for future applications to use such a feature would require no modifications in the AIREN network. Finally, when the LocalLink end-of-frame (EOF) signal is asserted, the routing module can deassert its request for the output port and the switch controller is free to reconfigure the connection to another input port.

*4.4. AIREN Switch Controller.* Once the routing decision has been made and the output port identified, the switch controller configures the crossbar switch to connect the input and output ports. By itself, the crossbar switch merely connects inputs to outputs with a single clock cycle of latency to register inputs to outputs. However, to control which inputs are connected to which outputs requires a switch controller. The AIREN network is configurable to support either a software controller or a hardware controller. The software controller enables the processor to set the connections within approximately 0.07$\mu$s. For systems with minimal changes to the communication path, the software-controlled switch is appealing as it offers low resource utilization. With each arriving packet, an interrupt is triggered to the processor to make the routing decision, which sets the connectivity of the input and output ports.

Of course, as the radix increases so too does the demand on the processor to make routing decisions. The switch can be controlled by hardware by connecting each input port to a routing module. Each routing module can make its own routing decision in parallel and notify the switch controller. The switch controller monitors each routing module and configures the switch based on input requests and output ports availability. Requests for separate output ports can be handled in parallel. To deal with contention, a simple priority encoder is used to give predetermined ports a higher priority. This mechanism can also be customized to support other arbitration schemes with minimal effort.

*4.5. On-Chip/Off-Chip Network Integration.* The architecture independent reconfigurable network (AIREN) card was designed in house to connect the nodes of the cluster. Each link in the direct connect network is a full-duplex high-speed serial line capable of transmitting/receiving at 8 Gbps (4 Gbps in/4 Gbps out). Each FPGA node in *Spirit* has eight links for a theoretical peak bandwidth of 64 Gbps to/from each node. The network was physically implemented by fabricating a custom network interface card (see Figure 5) that routes eight of the FPGA's integrated high-speed transceivers to SATA receptacles (we use SATA cables but not the SATA protocol). The AIREN network card has additional hardware used to manage the cluster, configure the FPGAs, and debug active designs.

The off-chip network is centered around the AIREN network card. Since each node can connect up to eight other nodes, a variety of network topologies are possible. The

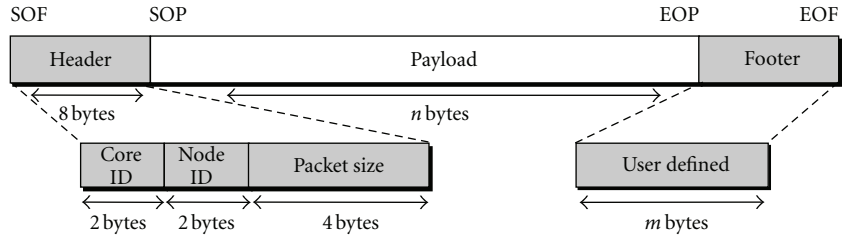



Figure 4: AIREN packet structure.

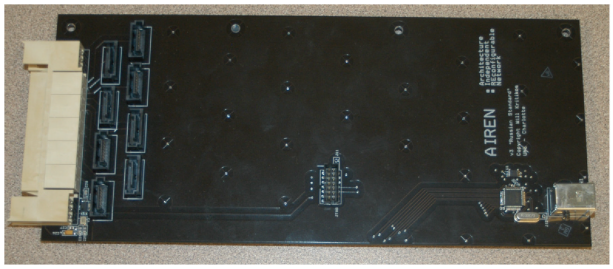



Figure 5: AIREN network card.

experiments reported here use a 4-ary 3-cube (four nodes in each of the three dimensions). Within the FPGA, each MGT is interfaced through the Xilinx Aurora protocol [26]. Aurora is a component released by Xilinx which wraps the multi-gigabit transceiver present on most of the new Xilinx FPGAs in an simple LocalLink interface. To this core the AIREN network adds custom buffers and flow control to more efficiently support back pressure between nodes. The Aurora core is configured to support bidirectional transfers of 32-bit at 100 MHz, resulting in a peak bandwidth of 4.00 Gbps. To this, Aurora uses 8B/10B encoding which further reduces the peak bandwidth to 3.2 Gbps.

*4.6. AIREN Resource Utilization.* There are three key parameters that can change the performance and resource requirements of our network core. First is the number of ports. This is the dominant parameter because, in general, the resources required to implement a full crossbar switch grow exponentially with the number of ports. The second parameter is the port bit width; it defines the number of bits transmitted in parallel during each clock cycle. The resources of the crossbar grow linearly with the number of bits per port. The third parameter is clock speed. Technically, all of the devices we consider have a maximum clock speed of 500 MHz; however, in practice, unless the design is hand optimized, frequencies between 100 and 200 MHz are typical. In Table 1 the resources consumed by a software-controlled and a hardware-controlled AIREN network. The purpose is to report on the scalability of the approach given the Virtex4 FX60 FPGA device resources. The key difference is the 4-input look-up table (4-LUT) utilization that is needed for the switch controller when performing hardware routing. In contrast, the software routing requires slightly more slice flip-flops (FFs) so the processor can interface with each output port configuration register.

When considering trends, it may be more instructive to increase the ports on the switch proportional to the growth in the number of logic resources. When considering different FPGA devices (different CMOS technologies), the resources needed to support a high-radix full crossbar switch on an FPGA dramatically decreases so much so that current Virtex 6 and emerging Virtex 7 devices make a full crossbar switch approach, such as AIREN, highly feasible. This is seen by fixing the switch to consume no more than 15% of the resources, as seen in Table 2. Since the behavior of the switch is very predictable and increasing the number of ports per switch will only reduce contention, this data suggest that the proposed approach is not just acceptable for the current generation of FPGA devices but also for the foreseeable technologies.

*4.7. AIREN Performance.* First, we test the network performance in isolation. Network layer messages between hardware and software are identical in the AIREN network. However, the ability of a core to generate or accept a message depends on the core. If a software task is the intended recipient, there is a significant amount of overhead (a processor interrupt, context switch, data, and more copies) compared to a typical hardware core (i.e., usually designed to produce or consume the message at network line speeds). There is also a speed difference if the message stays on chip versus off chip. This is because the network layer message is transmitted using a data link layer, Aurora, which handles transmission details (such as 8B/10B encoding, clock correction, etc.). Hence, we test five different combinations. The latencies are measured by performing a ping-pong test, measuring the round-trip time for a message to be transmitted from source to destination and then sent back. The round-trip time is then divided by two. The various latencies for a single hop are shown in Table 3. Using the hardware core to hardware core testing infrastructure, we looked at how the latency scaled with multiple hops as well. The results are summarized in Table 4. The results show that, as the number of hops increases, the latency per hop is decreasing. This suggests that the dimensionality of the network can be trusted to predict the overall latency of the network. This is an extremely important result because it speaks to the overall scalability of the design. (Note: all of these tests were conducted without any other jobs running on the machine; thus there was no outside contention.)

Table 1: AIREN resource utilization on V4FX60 FPGA.

| No. of ports | Software routing | | Hardware routing | |
|---|---|---|---|---|
| | No. of Slice FFs (%) | No. of 4-LUTs (%) | No. of Slice FFs (%) | No. of 4-LUTs (%) |
| 4 | 433 (0.85%) | 669 (1.32%) | 305 (0.60%) | 655 (1.30%) |
| 8 | 703 (1.39%) | 1,559 (3.08%) | 511 (1.01%) | 2,009 (3.97%) |
| 16 | 1,263 (2.49%) | 5,589 (11.05%) | 964 (1.91%) | 8,228 (16.27%) |
| 32 | 2,471 (4.89%) | 20,757 (41.05%) | 1,933 (3.82%) | 33,603 (66.46%) |

Table 2: Largest 32-bit full crossbar switch using 15% of device.

| Xilinx part | CMOS | Year | No. of ports |
|---|---|---|---|
| Virtex 2 Pro 30 | 90 nm | 2002 | 16 |
| Virtex 4 FX60 | 90 nm | 2004 | 20 |
| Virtex 5 FX130T | 65 nm | 2009 | 35 |
| Virtex 6 LX240T | 40 nm | 2010 | 70 |
| Virtex 7 855T | 28 nm | 2011 | 84 |

Table 3: Hardware send/receive latency through one hop.

| Sender/receiver pair | Latency ($\mu$s) |
|---|---|
| hw-to-hw (on chip) | 0.02 |
| hw-to-hw (off chip) | 0.80 |
| sw-to-hw (on chip) | 0.15 |
| sw-to-hw (off chip) | 0.98 |
| sw-to-sw (off chip) | 2.00 |

Table 4: Node-to-node latency with AIREN router.

| Hops | Latency | Latency/hop |
|---|---|---|
| 1 | 0.81 $\mu$s | 0.81 $\mu$s |
| 2 | 1.56 $\mu$s | 0.78 $\mu$s |
| 3 | 2.31 $\mu$s | 0.77 $\mu$s |
| 4 | 3.08 $\mu$s | 0.77 $\mu$s |

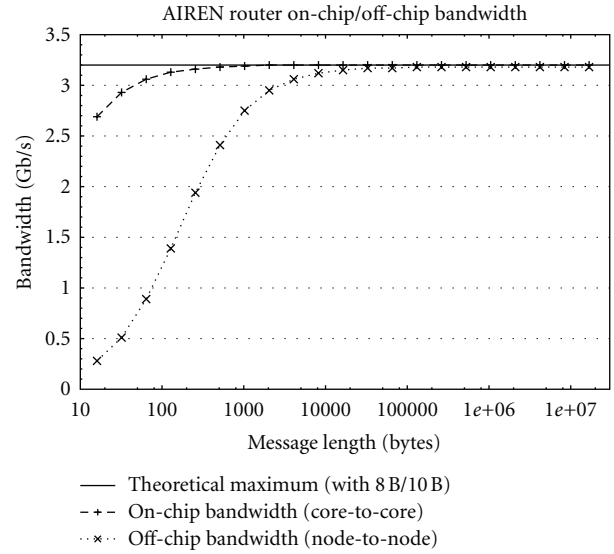



Figure 6: on-chip and off-chip bandwidth of AIREN network with hardware based routing.

To measure the bandwidth of the network, we varied the message length and repeated the ping-pong test. The measured bandwidth is plotted against the message length in Figure 6. Although the channel transmits at 4 Gbps, the data link layer performs 8B/10B encoding, a technique used to keep the sender and receiver synchronized. This, plus other communication protocol overhead, will decrease the effective data rate. Thus, after encoding, the peak theoretical bandwidth is 3.2 Gbps. Based on the data, messages on order of 10,000 to 100,000 bytes long approach optimal throughput. For on-chip bandwidth tests, messages are sent from one core to another core on the same node through the crossbar switch.

## 5. Accelerating Collective Communications

One of the advantages of implementing the networking cores in the FPGA fabric is the ability to rapidly introduce new hardware cores in the network data path. This allows offloading of general protocol processing [27] as well operations such as MPI collective communication operations [28]. Support for the standard MPI type collective communication tasks such as barrier, broadcast, and reduce can be added to any AIREN network implementation with the addition of a special-purpose core [29, 30]. These cores communicate among themselves using a programmable communication pattern and avoid interrupting the processor. Figure 14 shows how these cores can be connected to the AIREN router.

Specifically, the theory of operation is based on three key ideas. First, computation accelerator cores are not treated as "co-processors." Rather, they are first-class computation units that can receive and generate MPI-like messages in the system. This provides (i) a natural mechanism for processors and accelerator cores to operate in parallel and (ii) a simple performance model for computational scientists to follow. Second, the flexibility of the FPGA allows computer engineers to explore different ways of moving software functionality into hardware. For example, the active, decision-making components of the MPI collective communication operations are currently handled in the user-space MPI library. With the FPGA approach, this functionality can be moved into programmable logic as illustrated in Figure 7. Finally, by using FPGA-based compute nodes, we have the

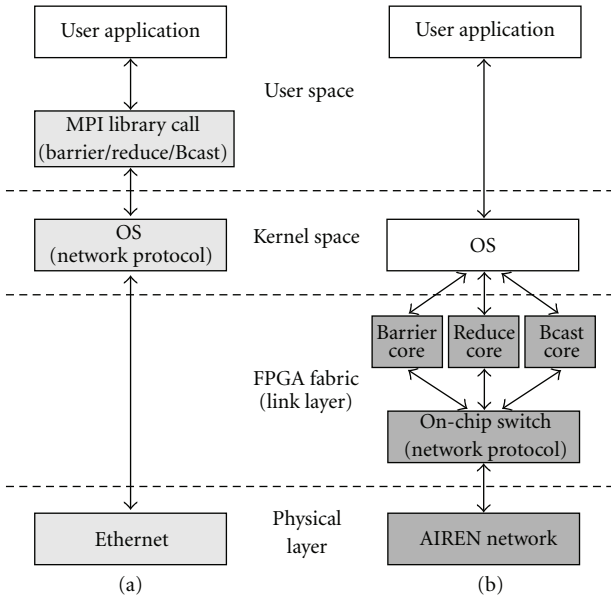



Figure 7: Example of system software refactored into hardware.

ability to tightly integrate on-chip and off-chip communication, which is the focus of this paper.

With the construction of an integrated on-chip and off-chip network operating at low latencies and high bandwidths, it became necessary to add support communication and compute accelerators. A drawback of a pure FPGA cluster is the slower clock rates of the processors (300 MHz PowerPC 405 on the Virtex4 FX60 FPGA). However, eliminating the processor entirely would drastically slow down development and reduce the chances for adoption due to the added complexities commonly associated with hardware IP development.

As a result, the RCC project chose to use the message passing interface (MPI) by replacing many of the MPI primitives with more efficient hardware-accelerated implementations. These include point-to-point operations, such as `MPI_Send` and `MPI_Recv`, as well as collective communication operations like `MPI_Barrier`, `MPI_Reduce`, and `MPI_Bcast`.

Each MPI accelerator core is connected to the AIREN network as well as the system bus. The processor is also connected to the system bus and can trigger the MPI operation through a few bus transactions needed to set up the operation. Once the MPI core is started, it performs any necessary DMA requests and communication associated with the operation.

*5.1. `MPI_Send` and `MPI_Recv` in Hardware.* For example, `MPI_Send` requires the processor to designate the receiving node, the base address of the data to be transmitted, and the size of the transfer. The core then issues the DMA request to off-chip memory to retrieve the data. As the data is received from off-chip memory, it is sent across the AIREN network as one or more packets to the destination. The destination receives the data and stores the data based on the corresponding `MPI_Recv` command. The tight integration between the memory controller and the network enables these transfers to occur with minimal involvement from the processor. Furthermore, the AIREN network does not require the data to be buffered and copied between the network and main memory.

Figure 8 illustrates how the hardware-based `MPI_Send` and `MPI_Recv` interact with the AIREN network. In the first step (1) the processor stores data in main memory to be sent to another node and (2) the processor initiates the DMA transfer indicating the destination node, database address, and size of the transfer (along with MPI communicator details). During the third step (3) the DMA core initiates the DMA transfer from (4) off-chip memory and (5) then generates and transfers the data to the AIREN router. The AIREN router then (6) transfers the data to the Aurora link layer core which will transfer (7) the packet to the receiving node's Aurora link layer core. Next, (8) the packet is then transferred to the destination node's router where it (9) routed to the DMA core. The DMA core interrupts (10) the destination processor, asking for a physical address (11) to store the data so that it can (12) store the data in off-chip memory. Finally, (13) the processor can use the data. The total time is dependent on the size of the transfer and the number of hops, but initial latency with respect to a single hop is shown in Table 3.

*5.2. `MPI_Barrier` in Hardware.* The easiest collective to implement is `MPI_Barrier`. Although many message-passing applications primarily synchronize by reasoning about the sequence of messages, there are few extremely important applications that heavily rely on barrier. For these applications, as the scale increases, the most important characteristic is reducing the time between the last task entering the barrier to when all tasks exit the barrier. Hardware is valuable here because it can typically propagate a message in a few cycles versus a software process which has a lot more overhead. There are several message-passing patterns that can be used to implement barrier (which have been explored in [29]).

Another example would be to synchronize processes or tasks executing in parallel. `MPI_Barrier` would typically be used to assure that all the tasks are operating in lockstep. A hardware-accelerated version of this core can perform all of the internal node-to-node communication without requiring any intervention from the processor, dramatically reducing the time for the barrier to complete. Four tree-based algorithms are used in this evaluation due to their low algorithm complexity and overall scalability: binomial, binary, start, and linear. Figure 9 reports the average time to complete a barrier for these topologies on the AIREN network.

*Binomial Tree.* This topology utilizes all the possible channels on each node. For example in a 4-ary 2-cube each node has 4 neighbor nodes directly connected. Theoretically, message transmissions can happen in all the channels in parallel, which could achieve the highest topology parallelism.

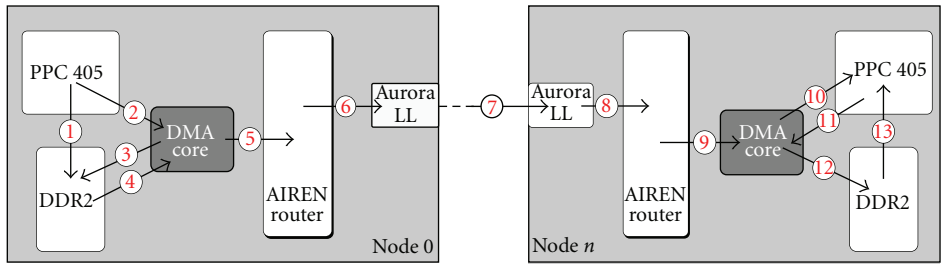



Figure 8: Example data flow of an AIREN hardware base `MPI_Send`/`MPI_Recv`.

*Binary Tree.* This topology is a subset of the binomial tree where each parent can only have at most two children. As the dimensionality of the network increases, this results in a reduction of the peak parallel performance as compared to the binomial tree.

*Star Tree.* This topology "virtually" connects the root node to all the other nodes. Messages hop through multiple nodes to the destination via the on-chip router. The benefit of this approach is that all nodes except the root node only need to act as children to the parent, simplifying the work done on each child node. However, when the number of nodes is large, the number of physical channels on the root node becomes the limiting factor causing contention and degrading the overall performance of the system.

*Linear Tree.* This topology has no parallelism. Every node has only one parent, and one child directly connected. Messages are relayed one node to another from the leaf to the root. This presents the worst case for barrier because the number of hops is linearly proportional to the size of the network.

*5.3. `MPI_Reduce` in Hardware.* The `MPI_Reduce` collective is a little more complicated than performing a barrier synchronization. The reduce core performs the specified reduce function in hardware, such as ADD or MAX, allowing the designer to take advantage of the parallelism within the FPGA when implementing each function [30]. In this approach, the reduce core receives its local data and fills the compute core's input pipeline. Each compute core then calculates its own local results. As results are produced, they are immediately transferred to the parent node. In fact, the tight integration between the computation and network creates one large pipeline between all of the nodes. Also, the only involvement from the processor is to begin the `MPI_Reduce` and provide a destination to store the final results. This significantly reduces the software overhead of traversing the network protocol stack.

When a reduce operation is initiated by the PowerPC (PPC), it tells the DMA engine what local data should be sent to the reduce core. The local data are fetched by the DMA engine and passed to the reduce core through the AIREN router. Depending on the role of the node (root, intermediates or leaf), the reduce core processes the data and transmits the result to the DMA on root or the parent node (intermediate or leaf). When the final result is sent to the DDR2 memory through the DMA engine, the reduce core interrupts the PPC indicating the reduce is complete.

Figure 10 overviews the reduce core's implementation on the FPGA, primarily comprised of three buffers, one or more compute cores, and the reduce control logic. Two input buffers are used for the compute core's input operands while the third buffer is for the computed results. The current implementation is designed to support double-precision floating-point data with a buffer size of 512 elements. Although the reduce core can be implemented with more FPGA resources to achieve better performance, the buffer size was selected to support the reduce operation of the processors (PowerPC or MicroBlaze) or custom hardware cores with minimal resource overhead, freeing up resources for other hardware accelerators or other collective communication cores. Furthermore, the reduce hardware core is operating at 100 MHz and is seriously handicapped compared to a commodity x86 class processor. If the reduction is to result in a single scalar value, the hardware core is very efficient. However, if the resulting reduction is a large vector, then it becomes a question of memory bandwidth and sequential computational speed.

The reduce core connects to the AIREN router through a single bidirectional connection which also reduces the resource utilization. As a result, the reduce core is designed to receive one message from the router at a time. To handle multiple incoming messages, FIFO_A is dedicated to store the data from local DMA transfers, FIFO_B stores the data from all of the children nodes, and FIFO_C stores the temporary result. FIFO_C can also feed the result back to the compute core if the node has more than two children. Likewise, for leaf nodes since no local computation is necessary (leaf nodes simply send their local data to their parents), FIFO_A is used to send data directly to the router and onto its parent.

While the compute core can be any associative operation, in these experiments, the compute core is a pipelined double-precision ADD unit. The ADD unit was chosen for its prevalence in a significant number of the `MPI_Reduce` function calls. The results presented in Figure 11 are more closely tied to the dataset size and the number of nodes rather than to the specific reduce operation being performed. By replacing the ADD unit with another compute core, only the difference in the latency across the compute core's pipeline will differ. The experiments are run over the four previously mentioned tree structures, and, as can be seen from the

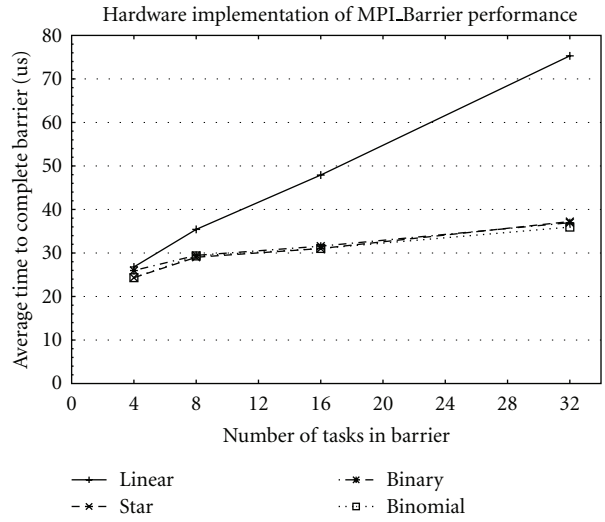



Figure 9: Measured `MPI_Barrier` performance of different network topologies on *Spirit*.

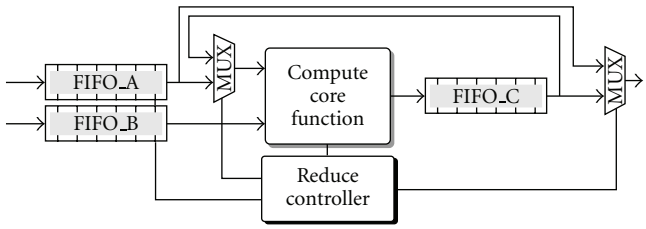



Figure 10: Block diagram of the hardware implementation of `MPI_Reduce`.

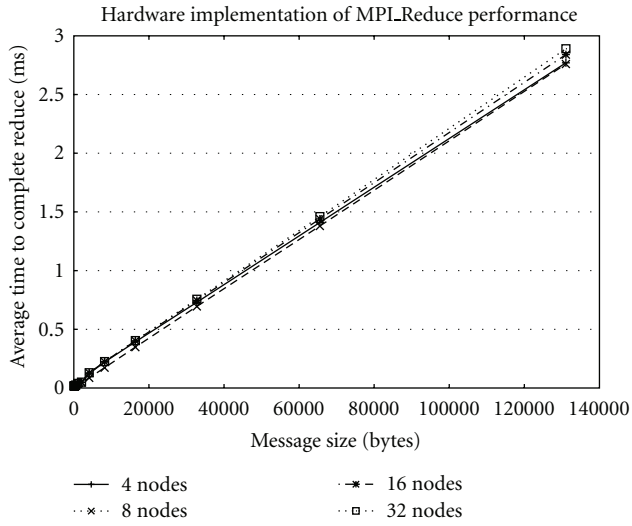



Figure 11: Measured `MPI_Reduce` performance of different network topologies on *Spirit*.

results, the average time to completion is less dependent on the tree structure than was found for barrier. Finally, the reason the hardware `MPI_Reduce` is linear is an artifact of our synthetic benchmark. A tight loop just performing reductions allows the hardware to organize its self into a deep pipeline with one task starting the next reduction before the previous one completely finished.

## 6. Integrating Compute Cores with AIREN

To demonstrate the capabilities of the AIREN network, a custom compute core was designed to perform matrix-matrix multiplication. Matrix-matrix multiplication can be decomposed into many multiply and accumulate (MAcc) steps. Figure 12 shows how the single-precision multiplier and adder are combined to form one multiply accumulate unit. The two inputs for the row and column values are connected to the inputs of the multiply unit. The result of the multiplication is one of the inputs to the adder. The other input to the adder is one of eight accumulation registers. The accumulation registers are necessary to hide the latency of the adder. The number of registers is proportional to the latency of the adder. If only one accumulation register was used, then the system would have to stall while waiting for the previous step's accumulation to finish. In this system the intermediate results of independent multiply-accumulate operations are being stored in the registers.

*Floating Point Unit.* While many FPGA-based floating point units have been presented in the literature [31–36], we chose to use the parameterizable floating point unit generated by the Xilinx CoreGen utility. The parameters of the floating point core include precision (single, double, or custom), utilization of DSP48 primitives in the FPGA, and latency (which affects both clock frequency and resource utilization). We wanted to maximize the number of multipliers and adders, while maintaining at least a 100 MHz clock frequency, use 100 percent of the DSP48 units, and approximately 50 percent of the LUT and FF units in the V4FX60 FPGA. We configured the multiply unit with a six clock-cycle latency, no DSP48 slices, 500 LUTs and 500 FF. The single-precision

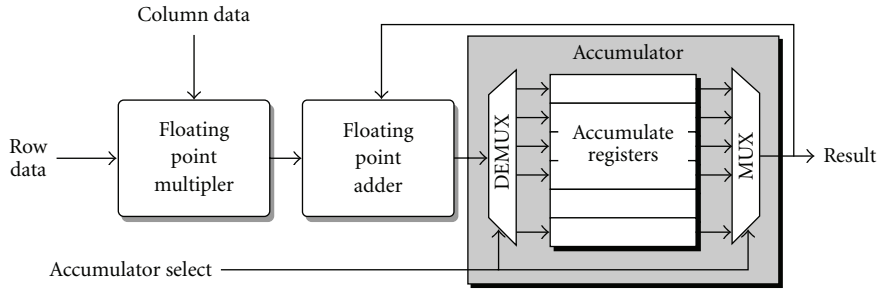



Figure 12: Single precision multiply accumulate unit.

adder was configured to have a seven clock-cycle latency and use four DSP48 slices, 500 LUTs and 500 FFs. These resources allowed us to instantiate up to 32 multipliers and 32 adders while using only half of the LUT and FF resources while maintaining the 100 MHz minimum system clock.

*MAcc Array.* Figure 13 shows MAcc units assembled into an array to support matrix-matrix multiplication. The FIFOs around the edge of the array help to keep new data available every clock cycle. The MAcc units are connected as an variable-sized array. On an FPGA the available resources limit the size of the array. For the Virtex-4 FX60 FPGA on the ML410 development board, there are 128 discrete processing slices (DPS). Single-precision floating point multiplication and addition can consume zero to several DSPs based on the desired operating frequency and latency. Therefore, a critical design decision is the size of the MAcc array.

*Application-Specific Memory Controller.* It has been shown that very low memory bandwidth is achievable for random access to off-chip memory from CCMs implemented in FPGA logic [37]. The MAcc array does not need random access to memory. We have developed a controller which uses burst transactions to off-chip DDR2 memory to request sequential blocks of data to fill the FIFOs in the MAcc array. This is a commonly overlooked advantage to building CCMs in FPGAs—you can often build a memory controller running in parallel to the computation unit which will request data in exactly the right order for the computation. This lessens or eliminates the need for a memory hierarchy and can fully utilize the relatively small off-chip memory bandwidth present in most FPGA-based systems. The FPGA system uses a multiport memory controller (MPMC) wrapper around the low-level DDR2 memory controller. The MPMC connects to the PowerPC processor and the MAcc array via a native port interface (NPI) connection. A custom controller was written to implement the NPI standard interface and fill the requested data into the row and column FIFOs.

*6.1. MAcc Array Integrated with AIREN.* Integrating the MAcc array with AIREN includes a processor for control and coordination, off-chip memory controller to provide access to the large matrices to be accessed by the application specific memory controller, and the network IP cores for AIREN. Figure 14 illustrates this configuration, along with the MPI hardware accelerator cores that can be used to support `MPI_Send/Recv`, `MPI_Barrier`, `MPI_Broadcast`, and MPI_Reduce. The base architecture includes all but the hardware accelerator core, in this example the MAcc array. Development begins with this initial infrastructure already in place. Therefore, the designer needs only to integrate the new core within this construct, saving significant development time since the base system is already functionality in place.

While early efforts focused on the use of Ethernet prior to the development of AIREN, the performance bottleneck of the Ethernet network was far too limiting. Instead, AIREN, provides a significantly greater bandwidth and much tighter integration between the hardware-accelerator cores and off-chip memory. The use of the AIREN network does require additional resources that are not necessary with a traditional Ethernet approach. In this study the initial Ethernet design used a $16 \times 2$ (almost fully utilized FPGA resources) MAcc array, but used Fast Ethernet instead of the AIREN network. The single node performance yielded $\approx 3.8$ GFLOPS; however, as the system was implemented on the *Spirit* cluster, the Ethernet network quickly limited the performance as the system scaled. A significant amount of time was spent by each node waiting for the transfers of data between nodes to complete.

To include the AIREN network required reducing the size of the MAcc array, from $16 \times 2$ to $8 \times 2$. This meant the peak performance would drop from 6.4 GFLOPS to 3.2 GFLOPS (100 MHz clock frequency). Ultimately, the sacrifice in performance was justified based on the insufficient performance provided by fast Ethernet. Comparable tests were run for the $8 \times 2$ MAcc array and the performances are presented in Figure 15.

In this implementation, all network transfers are performed across the AIREN's high-speed network. Transfers are also performed in parallel (for both matrix A and matrix B). This is due to the fact that each node has two DMA cores and can be performing an `MPI_Send` and `MPI_Recv` on each matrix in parallel. Furthermore, the hardware implementation of `MPI_Barrier` is used to speedup synchronization between nodes. Overall, the results show significant performance gains even over a much larger 49 node implementation with twice the computed resources ($16 \times 2$ versus $8 \times 2$). Certainly it makes sense that,

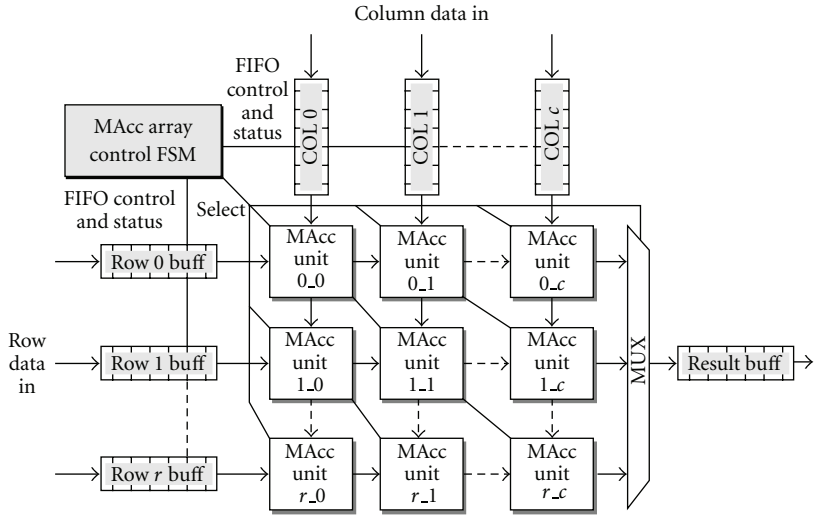



Figure 13: A variable sized array of MAcc units.

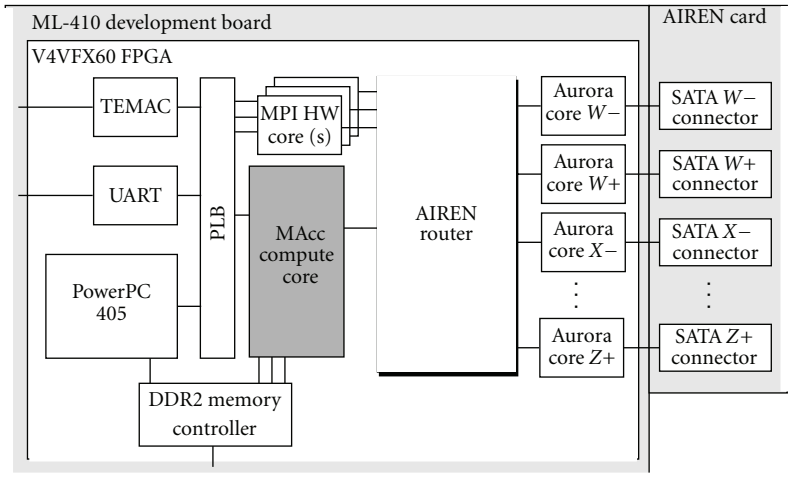



Figure 14: Single ML410 base system configuration for MAcc Array with AIREN.

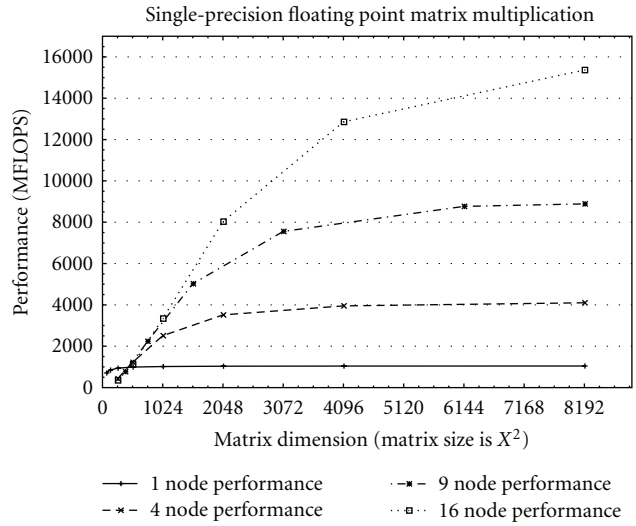



Figure 15: Performance of the 8 × 2 MAcc array implemented on a single, 4, 9, and 16 nodes of the Spirit cluster with the AIREN network.

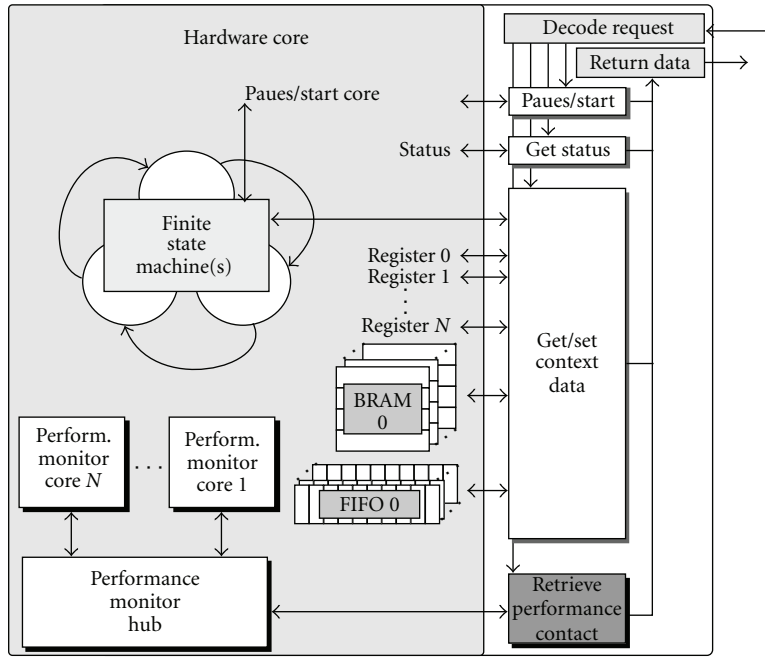



Figure 16: Block diagram of hardware core and its performance monitoring infrastructure.

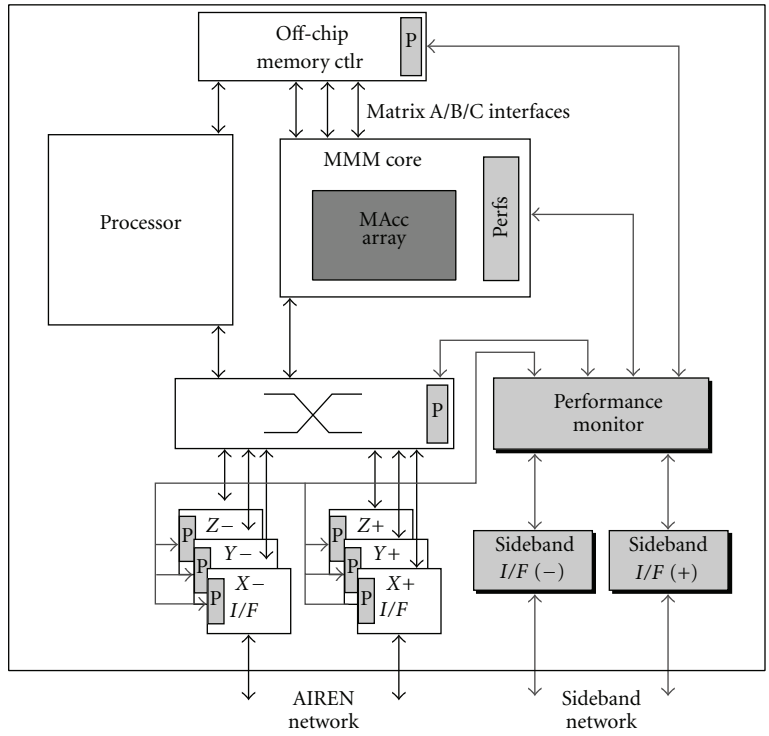



Figure 17: Block diagram of the performance monitor infrastructure used in the matrix-matrix multiplication evaluation.

when transferring large datasets between nodes, the higher bandwidth AIREN implementation will outperform Fast Ethernet. However, for a designer to realize the need to sacrifice single node performance to include the necessary on-chip infrastructure for the network might not be so easily identifiable.

## 7. Performance Monitoring with AIREN

Another unique aspect of the AIREN on-chip/off-chip network is that it can be used to support a sideband network independent of the primary data network. Recent work [38, 39] has begun to explore using this sideband network to

monitor multichip system performance. Because the monitoring is handled in hardware, it has the ability to probe more extensively and less invasively than a processor core that uses the primary network. Capabilities provided by this network include the ability to retrieve status (health), get/set context (state), and performance (detect stalls, e.g.). The motivation for this research and its integration into the existing AIREN network is that in high-performance computing, as the number of tasks increases it becomes increasingly difficult to quickly and efficiently perform basic capabilities like checkpoint/restart and is even more complicated to identify whether the system is making progress towards a solution.

The ability to retrieve status information and get/set context may improve system resiliency; however, we envision a system that is more proactive. By monitoring the performance of hardware cores and other key components (channel utilization, rate of interrupts, how often FIFOs are empty, etc.), we hypothesize that it is possible to characterize and more accurately predict when failures will occur as well as determine if useful work is being performed. Of course, the increasing amount of performance data per node requires an efficient network. Current research uses the two remaining network channels on the AIREN network to form a sideband network as a ring through the cluster. Performance monitoring data can be passed between nodes without interfering with the primary AIREN network.

To accomplish this, we have integrated a performance monitoring infrastructure (consisting of a monitor hub and monitor cores) into the existing monitoring system to allow a separate monitoring node access to the performance information without any additional overhead on the cluster's compute nodes. The *performance monitor hub* is instantiated within the hardware core, an example of which can be seen in Figure 16. The purpose of the hub is to collect the performance data and send it to the monitoring node for future evaluation. The sampling period can be adjusted down to microsecond intervals, depending on the amount of data to be sent per node.

A *performance monitor core* connects to the performance monitor hub and performs a specific monitoring function. Examples include timers and counters, to more complicated pattern detectors, utilization and efficiency monitors, and histogram generators. These, and many more, are regularly added to hardware cores (a convenient feature FPGAs offer to the HPC community) by a designer. We have created a repository of these monitor cores that, with this performance monitoring infrastructure, can be easily added to a design. Since many designs do not fully utilize all of the FPGA resources, the cost is often negligible. For a designer to see runtime information at such a low level (without introducing interference to the system and processor) will also result in more efficient designs.

While analyzing the performance of the $8 \times 2$ MAcc Array, it became clear that even on a single node the effective performance was approximately a third of the theoretical peak performance. Some of the performance loss can be attributed to software overhead; however, in order to fully understand the performance sink, the performance monitoring system was integrated into the system. For the purposes of this experiment, the MAcc Array, memory controller interfaces, and network interfaces were monitored with the intention identify to which core(s) was under performing. Figure 17 illustrates how the performance monitoring infrastructure is incorporated into the existing matrix-matrix multiplication design and with the AIREN network.

From this runtime performance data, it was identified that the custom memory controller to read the matrices from off-chip memory was severely underutilized. In effect, the memory controller was obtaining ≈32% of the available bandwidth (0.53 GB/s out of 1.6 GB/s). The custom memory interface performs requests per row; however, the row size was modified from 16 to 8 without redesigning the memory controller interface. Furthermore, the utilization of the on-chip buffers within the MAcc array were underutilized, often only storing as much as a single row of the submatrix. With a more efficient memory interface, this utilization will increase; however, a resource tradeoff could be made by reducing the size of the buffers to the largest submatrix to be evaluated.

## 8. Conclusion

*8.1. Summary.* The integrated on-chip/off-chip network developed under the RCC project was originally proposed in 2005. It has changed considerably since then, largely because application development informed our decisions. As this paper reports, the network has proven to be general enough to cover a wide range of applications (from MPI point-to-point messages to system-wide monitoring) while maintaining scalability.

Perhaps the most striking difference between the network architecture presented here and NoC literature is the use of a single full crossbar switch. Our results are different from other efforts due to several reasons. First, our target is the programmable logic of an FPGA not CMOS transistors, and this affects how one writes the hardware description language. As a result, we achieve much better resource utilization. Also, because our target frequency is much lower (100s of MHz) instead of Gigahertz, the physical distance that a signal can travel in one clock period is different. Specifically, our "reach" is in most cases the entire chip whereas future gigahertz CMOS chips might only be able to reach a fraction of the way across a chip. Second, most NoC assumes that a full crossbar is too expensive in terms of resources. However, within the context of a HPC system, the question is "what is too expensive?" In the case of an FPGA, the programmable logic resources are fungible. In most of the HPC applications we have worked with, the compute accelerator cores provide an enormous amount of computation. Overall performance is limited by the bandwidth off-chip so adding additional compute cores does not improve performance. In that circumstance, it makes sense to use as much of the available resources as possible to increase the network performance.

*8.2. Future Work.* We are presently designing a replacement for *Spirit*. It will likely use Virtex 7 devices that have 36

10 Gbps transceivers organized into a 5-ary 4-cube where each link is four-bonded transceivers yielding a bisection bandwidth of 20 Tbps. Even though the number of FPGAs is scaled from 64 to 625, the impact on our network design is minimal. This is because we are going from six network ports to eight network ports on the crossbar. In contrast, because of Moore's law, the size of a single FPGA will grow dramatically. Our Virtex 4 devices have around 57,000 logic cells; the Virtex 7 will likely have between 500,000 and 750,000 logic cells.

## References


[1] A. G. Schmidt, W. V. Kritikos, R. R. Sharma, and R. Sass, "AIREN: a novel integration of on-chip and off-chipFPGA networks," in *Proceedings of the 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '09)*, April 2009.

[2] B. Sethuraman, P. Bhattacharya, J. Khan, and R. Vemuri, "LiPaR: A light-weight parallel router for FPGA-based networks-on-chip," in *Proceedings of the ACM Great Lakessymposium on VLSI (GLSVLSI '05)*, pp. 452–457, April 2005.

[3] S. Kumar, A. Jantsch, J. Soininen et al., "A network on chiparchitecture and design methodology," in *Proceedings of the IEEE Computer Society Annual Symposium on (VLSI '02)*, vol. 102, pp. 117–124, 2002.

[4] R. Gindin, I. Cidon, and I. Keidar, "NoC-based FPGA: architecture and routing," in *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS '07)*, pp. 253–262, May 2007.

[5] G. Schelle and D. Grunwald, "Exploring FPGA network on chip implementations across various application and network loads," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 41–46, September 2008.

[6] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Proceedings of the 38th Design Automation Conference*, pp. 684–689, June 2001.

[7] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys*, vol. 38, no. 1, pp. 71–121, 2006.

[8] I. Walter, I. Cidon, A. Kolodny, and D. Sigalov, "The era of many-modules soc: revisiting the noc mapping problem," in *Proceedings of the 2nd International Workshop on Networkon Chip Architectures*, ACM, 2009.

[9] G. Michelogiannakis, D. Sanchez, W. J. Dally, and C. Kozyrakis, "Evaluating bufferless flow control for on-chip networks," in *Proceedings of the 4th ACM/IEEE International Symposium on Networks on Chip (NOCS '10)*, pp. 9–16, May 2010.

[10] R. Sass, W. V. Kritikos, A. G. Schmidt et al., "Reconfigurable Computing Cluster (RCC) Project: investigating the feasibility of FPGA-based petascale computing," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*, IEEE Computer Society, April 2007.

[11] P. Kogge et al., "Exascale computing study: technologychallenges in achieving exascale systems,," Tech. Rep. TR-2008-13, DARPA Information Processing Techniques Office (IPTO), 2008, http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf.

[12] K. D. Underwood, W. B. Ligon III, and R. R. Sass, "An analysis of the cost effectiveness of an adaptable computing cluster," *Cluster Computing*, vol. 7, pp. 357–371, 2004.

[13] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[14] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, *Splash2: FPGAs in a Custom Computing Machine*, Wiley-IEEE Computer Society Press, 1996.

[15] D. Burke, J. Wawrzynek, K. Asanovic et al., "RAMP blue: Implementation of a manycore 1008 processor system," in *Proceedings of the Reconfigurable Systems Summer Institute*, 2008.

[16] C. Pedraza, E. Castillo, J. Castillo et al., "Cluster architecture based onlow cost reconfigurable hardware," in *Proceedings of the International Conferenceon Field-Programmable Logic and Applications*, pp. 595–598, IEEE Computer Society, 2008.

[17] M. Saldana and P. Chow, "TMD-MPI: an MPI implementationfor multiple processors across multiple FPGAs," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pp. 1–6, IEEE Computer Society, 2006.

[18] A. P. Michael Showerman and J. Enos, "QP: a heterogeneousmulti-accelerator cluster," in *Proceedings of the International Conferenceon High-Performance Cluster Computing*, 2010.

[19] P. P. Kuen Hung Tsoi, A. Tse, and W. Luk, "Programming framework for clusters with heterogeneous accelerators," in *Proceedings of the International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2010.

[20] NSF Center for High Performance Reconfigurable Computing (CHREC), "NOVO-G: adaptivelycustom research supercomputer," April 2005, http://www.xilinx.com/support/documentation/sw manuals/edk92i ppc405 isaext guide.pdf.

[21] RAMP, "Research accelerator for multiple processors," August 2008, http://ramp.eecs.berkeley.edu.

[22] R. Baxter, S. Booth, M. Bull et al., "Maxwell—a 64 FPGA supercomputer," in *Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS '07)*, pp. 287–294, August 2007.

[23] Xilinx, "ML410 Development Platform—User's Guide UG085," 2007.

[24] Y. Rajasekhar, W. V. Kritikos, A. G. Schmidt, and R. Sass, "Teaching FPGA system design via a remote laboratory facility," in *Proceedings of the 18th Annual Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 687–690, IEEEComputer Society, September 2008.

[25] Xilinx, "Local Link Interface Specification SP006 (v2.0)," July 2005.

[26] Xilinx, "LogiCORE IP Aurora v3.0 Users Guide 61," 2008.

[27] R. G. Jaganathan, K. D. Underwood, and R. Sass, "A configurable network protocol for cluster based communications using modular hardware primitives on an intelligent NIC," in *Proceedings of the 11th Annual IEEE Symposiumon Field-Programmable Custom Computing Machines (FCCM '03)*, IEEE Computer Society, 2003.

[28] K. D. Underwood, *An evaluation of the integration ofreconfigurable hardware with the network interface in clustercomputer systems*, Ph.D. thesis, Clemson University, August 2002.

[29] S. Gao, A. G. Schmidt, and R. Sass, "Hardware implementation of MPI barrier on an FPGA cluster," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 12–17, August 2009.

[30] S. Gao, A. G. Schmidt, and R. Sass, "Impact of reconfigurable hardware on accelerating MPI_reduce," in *Proceedings of the*

*International Conference on Field Programmable Technology (FPT '10)*, IEEE Computer Society, December 2010.

[31] J. Liang, R. Tessier, and O. Mencer, "Floating point unitgeneration and evaluation for fpgas," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 185–194, April 2003.

[32] S. Chen, R. Venkatesan, and P. Gillard, "Implementation of Vector Floating-point processing Unit on FPGAs for high performance computing," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE '08)*, pp. 881–885, May 2008.

[33] W. Ligon III, S. McMillan, G. Monn et al., "A re-evaluation of the practicality of floating-point operations on fpgas," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, p. 206, IEEE Computer Society, Washington, DC, USA, 1998.

[34] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert, "Embedded floating-point units in FPGAs," in *Proceedings of the 14th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '06)*, pp. 12–20, February 2006.

[35] K. S. Hemmert and K. D. Underwood, "An analysis of the double-precision floating-point FFT on FPGAs," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 171–180, April 2005.

[36] X. Wang, S. Braganza, and M. Leeser, "Advanced componentsin the variable precision floating-point library," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 249–258, IEEE Computer Society, Los Alamitos, Calif, USA, 2006.

[37] A. G. Schmidt and R. Sass, "Characterizing effective memory bandwidth of designs with concurrent High-Performance Computing cores," in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 601–604, Amsterdam, The Netherlands, August 2007.

[38] B. Huang, A. G. Schmidt, A. A. Mendon, and R. Sass, "Investigating resilient high performance reconfigurable computing with minimally-invasive system monitoring," in *Proceedings of the 4th International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA '10)*, IEEE Computer Society, November 2010.

[39] A. G. Schmidt, B. Huang, and R. Sass, "Checkpoint/restart and beyond: resilient high performance computing with FPGAs," in *Proceedings of the 19th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*, IEEE Computer Society, May 2011.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 163542, 17 pages
doi:10.1155/2012/163542

*Research Article*

# A Coarse-Grained Reconfigurable Architecture with Compilation for High Performance

**Lu Wan,[1] Chen Dong,[2] and Deming Chen[1]**

[1] *ECE Illinois, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2918, USA*
[2] *Magma Design Automation, Inc., San Jose, CA 95110, USA*

Correspondence should be addressed to Deming Chen, dchen@illinois.edu

Received 5 October 2011; Revised 5 January 2012; Accepted 9 January 2012

Academic Editor: Kentaro Sano



We propose a *fast data relay* (FDR) mechanism to enhance existing CGRA (*coarse-grained reconfigurable architecture*). FDR can not only provide multicycle data transmission in concurrent with computations but also convert resource-demanding inter-processing-element global data accesses into local data accesses to avoid communication congestion. We also propose the supporting compiler techniques that can efficiently utilize the FDR feature to achieve higher performance for a variety of applications. Our results on FDR-based CGRA are compared with two other works in this field: ADRES and RCP. Experimental results for various multimedia applications show that FDR combined with the new compiler deliver up to 29% and 21% higher performance than ADRES and RCP, respectively.

## 1. Introduction and Related Work

Much research has been done to evaluate the performance, power, and cost of reconfigurable architectures [1, 2]. Some use the standard commercial FPGAs, while others contain processors coupled with reconfigurable coprocessors (e.g., GARP [3], Chimaera [4]). Meanwhile, *coarse-grained reconfigurable architecture* (CGRA) has attracted a lot of attention from the research community [5]. CGRAs utilize an array of pre-defined *processing elements* (*PEs*) to provide computational power. Because the PEs are capable of doing byte or word-level computations efficiently, CGRAs can provide higher performance for data intensive applications, such as video and signal processing applications. In addition, CGRAs are coarse grained so they have smaller communication and configuration overhead costs compared to fine grained field programmable gate arrays (FPGAs).

Based on how PEs are organized in a CGRA, the existing CGRAs can be generally classified into linear array architecture and mesh-based architecture. In linear array architecture, PEs are organized in one or several linear arrays. Representative works in this category are RaPiD [6] and PipeRench [7]. RaPiD can speed up highly regular, computational intensive applications by deep pipelining the application on a chain of RaPiD cells. PipeRench provides speedup for pipelined application by utilizing PEs to form reconfigurable pipeline stages that are then interconnected with a crossbar.

The linear array organization is highly efficient when the computations can be linearly pipelined. With the emergence of many 2D video applications, the linear array organization becomes less flexible and inefficient to support block-based applications [8]. Therefore, a number of mesh-based CGRAs are proposed. Representative works in this category include KressArray [9], MATRIX [10], and MorphoSys [8].

The KressArray was one of the first works that utilized 2D mesh connection in CGRA. KressArray uses *reconfigurable data path units* (rDPUs) as basic computation cell. These rDPUs are connected through local *nearest neighbor (NN)* links and global interconnection [9]. To accommodate high volume of communications between function units, the MATRIX [10] proposed a 3-level network connecting its *basic function unit* (BFU) including NN connection, length-4 bypassing connections and global lines. Finally, Morphosys [8] used an $8 \times 8$ PE array, divided into four $4 \times 4$ quadrants, as the coprocessor for a RISC host CPU. Each PE is directly connected with any PE in the same row/column within

the same quadrant and its NN regardless of the quadrants. Morphosys also exploits a set of buses called *express lanes* to link PEs in different quadrants.

Two interesting CGRAs are recently proposed: ADRES [11] and RCP [12]. ADRES belongs to mesh-based architecture. It utilizes Morphosys' communication mechanism and resolves resource conflict at compile time using modulo scheduling. RCP belongs to linear array architecture. It proposes an architecture with ring-shaped buses and connection boxes. One unique feature of RCP is that it supports concurrent computation and communication [12]. However, its hardware and software only support this feature in one dimension, which may limit RCP performance. Another representative architectural work is RAW [13], which pioneered the concept of concurrent computation and communication using MIPS processor array. However, due to the complexity of MIPS processor as a PE, RAW may not be an efficient platform to accelerate kernel code used in embedded applications. On the compiler side, a recent work is SPR [14], which presents an architecture-adaptive mapping tool to compile kernel code for CGRA. Their compiler targets an FPGA-like CGRA, and it supports architectures with a mix of dynamically multiplexed and statically configurable interconnects.

Some recent studies [15–18] show that further exploiting instruction level parallelism (ILP) out of ADRES architecture can cause hardware and software (compiler) inefficiency. This is mainly because ADRES utilizes heavily ported global register file and multi-degree point-to-point connections [18]. In particular, to implement communication pattern that broadcasts shared data on ADRES is challenging [15]. Using integer linear programming to find better code schedule may incur prohibitive long runtime [16]. Given the difficulty of further pushing for ILP performance, [17] turned to exploit thread-level parallelism in MT-ADRES.

In this work, to exploit ILP performance further, we propose a new mechanism, named *fast data relay* (*FDR*), in CGRA. This proposed FDR architecture replaces the heavily ported global register file with distributed simple register files (Section 2.2). It also replaces the point-to-point connections with simple wire-based channels (Section 2.3). The main features of FDR include (1) fast data links between PEs, (2) data relay where communication can be done in multiple cycles as a background operation without disturbing computations, and (3) source operand that can have multiple copies with longer lifetimes in different PEs so that a dependent PE can find a local copy in its vicinity. We name the proposed new architecture *FDR-CGRA*. These FDR mechanisms are generic and could also be incorporated into ADRES architecture to enhance the ILP performance. To utilize FDR efficiently, we propose new compiler techniques to map kernel code onto FDR-CGRA leveraging its unique hardware features.

In the following, we provide the motivation and the architecture support of FDR-CGRA in Section 2. In Section 3, we introduce our CAD-based compiler support for the efficient use of FDR. The experimental results will be discussed in Section 4, and we conclude this paper in Section 5.

## 2. Motivation and Architectural Support for *FDR*

With the shrinking of CMOS technology, more and more devices can be integrated into a single die. The granularity of CGRA has also gradually been raised from 4 bit to 16 bit or even 32 bit. The ADRES uses 32 bit ALU as its basic computation unit and is one of the representative modern CGRA architectures for mesh-based designs. It couples a VLIW processor with an array of PEs. Each PE consists of a predicated ALU, local register file, and I/O multiplexers. The functionality of each ALU can be configured dynamically through the *context SRAM* (CS). A unified *global register file* (GRF) is used to provide data exchange among PEs and the host VLIW processor. The interconnection of ADRES exploits the communication mechanism similar to Morphosys. Its compiler uses modulo scheduling algorithm to resolve resource conflict at compile time.

The research in [18] extensively studied how the configuration of register files impacts the performance of ADRES. It showed that the unified GRF is a severe performance bottleneck. In ADRES, each PE connected to the GRF has dedicated read and write ports to the GRF. In a 64-PE ADRES baseline configuration organized as 8 rows and 8 columns, eight PEs on the top row are connected to the GRF. The study in [18] showed that even this baseline configuration needs a highly complicated 8-read and 8-write GRF with significant area penalty. And at the same time, allowing only 8 simultaneous reads and 8 simultaneous writes to this GRF seriously limited the performance (IPC < 14) of 64-PE ADRES. It is also shown in [18] that to achieve an IPC above 23 on the original 64-PE ADRES architecture, the global register file needs to support >40 sets of read and write ports, which is, if not impossible, very challenging and costly for physical implementation. Also, their study pointed out that increasing the number of registers in GRF and each PE local register file can only contribute to performance marginally. Due to this limitation, the performance of ADRES can barely take advantage of the growing budget given by ever-shrinking modern CMOS technology.

RCP is a representative architecture for linear array-based design. But linear array architecture may not be able to accommodate block-based applications as well as mesh-based designs. To study this, we will also compare our results with RCP. In the rest of this section, we propose several architectural enhancements for the mesh-based architecture to improve the performance. Then, in the section afterwards, we will propose several compiler techniques for better utilizing these new hardware features. As will be shown in the experimental result section, these hardware changes as well as the new compiler techniques can exploit higher instruction level parallelism (ILP) comparing to ADRES and RCP.

The architectural changes are summarized into a single term named fast data relay (FDR), which includes three main features: *bypassing registers*, *wire-based companion channels*, and the use of *non-transient copies*. In the following subsections, we will first introduce the overall architecture of FDR-CGRA. Then, each of the aforementioned three features will be explained in detail, respectively.

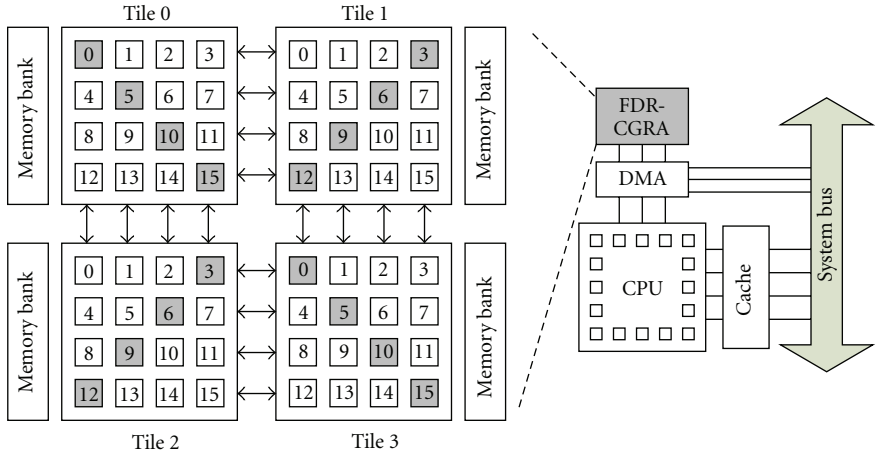



Figure 1: FDR-CGRA system overview.

*2.1. Architectural Overview.* Similar to existing reconfigurable computing platforms surveyed in [2], we assume that a host CPU initializes the kernel computation on our FDR-CGRA and is in charge of system-level dataflow via DMA. We assume filling data via DMA can be done in parallel with CGRA computations using double buffers (Ping-Pong buffers). Thus, FDR-CGRA can focus on accelerating time-consuming kernels. Figure 1 provides a conceptual illustration of the overall FDR-CGRA system. In FDR-CGRA, the basic computation units are PEs (Section 2.2) organized in tiles, and a wire-based communication layer (Section 2.3) glues PEs together.

FDR-CGRA adopts the tile-based design philosophy, like that of existing commercial FPGAs, for design scalability. Considering that memory banks have limited bandwidth, we restrict that only the shadowed PEs can access memory banks with load/store instructions (Figure 1).

To exploit more instruction level parallelism, it is desirable to exchange data for operations on non-critical paths without intervening with critical computations. Fast data relay is such a mechanism. It is capable of routing data from one PE to another in multiple cycles in parallel with computations by utilizing the bypassing registers (Section 2.2) and companion channels (Section 2.3). It is an enhancement over ADRES point-to-point communication. The philosophy of concurrent computation and communication was exploited in the RAW microprocessor [13]. However, instead of using switches that route data dynamically, FDR-CGRA uses wire-based channels. Communication within a tile is faster in FDR-CGRA than in RAW. The detailed analysis can be found in Section 2.3.

*2.2. Processing Element and Bypassing Registers.* As shown in Figure 2, each PE has a dedicated computation data path (left) similar to the PE used in ADRES and a communication bypassing path (right) that can be used for FDR. This dedication is essential to enable simultaneous computation and communication. Similar to ADRES, each ALU is driven by traditional assembly code and its functionality can be configured dynamically through the *context SRAM* (*CS*). The CS stores both the function specification of the ALU and the communication information that provides cycle-by-cycle reconfiguration of channels among PEs. The communication information includes control bits for the input multiplexers (1, 2, and B) on the top of the PE and output multiplexers at the bottom (3 and 4).

One key hardware feature of FDR-CGRA is the use of distributed bypassing register file (marked as BR in Figure 2): in addition to the normal local register file (marked as LR), which is used to store intermediate data for ALU, our PE includes a bypassing register file, through which multicycle data communication can be carried out. A leading PE can deposit a value in the bypassing register of a leading PE and a trailing PE can fetch it later on without disturbing computations on the leading PE. Such multicycle data communication is an effective way to reduce communication congestion, because it can detour communication traffic for non-critical operations to uncongested area. Note that the way our compiler utilizes the bypassing registers is different from the way the PipeRench compiler utilized its passing registers. In PipeRench, the passing registers were solely used to form virtual connections when a connection needs to span multiple stripes. And due to the greedy nature of the PipeRench compiler [19], whenever a path is unroutable due to resource conflict, a NOOP is inserted as the solution. As we will point out in Section 3.2 and in the experimental results, our compiler will use the bypassing registers as a powerful resource to solve routing congestions and exploit more ILP from application.

The bypassing register file associated with each PE provides a way for information exchange among PEs without incurring the same design complexity as in ADRES. As pointed out in [20, 21], central register file with a large number of read/write ports involves considerable complexity from multiplexing and bypassing logic, resulting in area penalty and timing closure challenges. The authors of [20] advocated to use distributed register files for media processing instead of using a heavily ported monolithic central

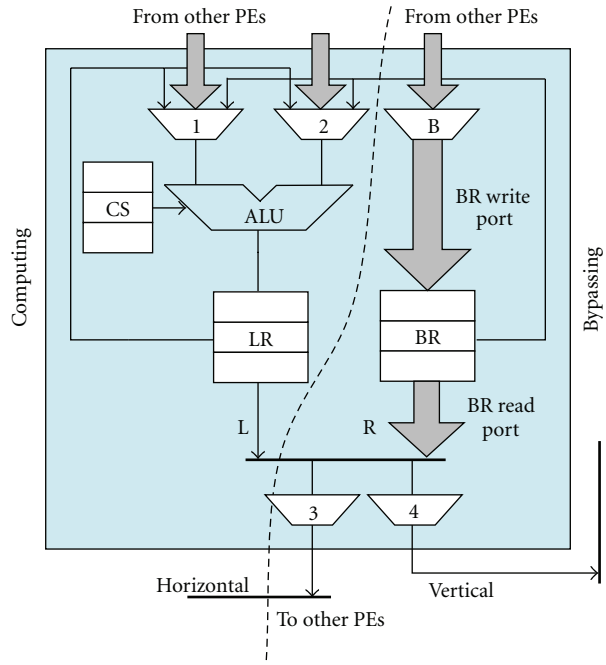



Figure 2: Architectural support for fast data relay at the PE level.

register file. Following this approach, in our FDR-CGRA, the distributed bypassing registers replace the unified GRF used in ADRES and its descendants [11, 15, 22]. To further simplify the register file design, we restrict the number of simultaneous accesses to the bypassing register file to 2 reads and 2 writes (2R2W). This constraint is explicitly modeled, and our tool can honor it during operation scheduling, mapping, and routing. By distributing the unified data storage in ADRES into small pieces as bypassing registers, we can exploit more ILP out of applications and gain larger programming flexibility over the original ADRES. However, at the same time, they pose some challenges for our compiler. This will be discussed in Section 3.

*2.3. Wire-Based Companion Channels.* In FDR-CGRA, inter-tile communication is simply through direct connections between PEs on the tile boundaries, as shown in Figure 1, while intratile communication is implemented with companion channels (Figure 3). Each PE owns two private companion channels: a vertical channel and a horizontal channel. Within a tile, the owner PE can send data via its private vertical (horizontal) channel to any PE in the same column (row) in a time-multiplexing way. Note that a vertical/horizontal channel can only be used by its owner for sending (not receiving) data. In this sense, these vertical (horizontal) channels are *unidirectional*. In Figure 3, for example, the companion channels owned by PE12 are PE12V and PE12H. All other PEs have similar sets of companion channels: there are four *unidirectional* vertical channels for each column of PEs and four *unidirectional* horizontal channels for each row of PEs. Note that vertical/horizontal channels span only across the PEs within the same tile for scalability reasons. Comparing to the 3-level interconnections used in Morphosys and ADRES where each PE has a set of dedicated links connecting to all PEs in the same row and column within a quadrant, our organization of companion channels is simpler because each PE only has one vertical and one horizontal channel and both are unidirectional. Initialized by the sender PE, scalar data can be deposited onto either horizontal or vertical channel through the output multiplexer. In the same cycle, the receiver PE can fetch the scalar data from its corresponding input multiplexer. The control bits of the output multiplexer and input multiplexer are determined at compile time and stored in the Context SRAM.

Comparing with multipoint point-to-point connection used in other tile-based architecture [8, 11, 15], utilizing the proposed simple wire-based companion channels may have some benefits when it comes to area cost. Firstly, the regularity of these proposed companion channels makes them easier to be implemented on top of PEs in metal layers without complicated routing to save routing area. Secondly, for intra-tile communication, the proposed companion channels can actually save configuration bits. Multi-point point-to-point interconnection needs several bits to specify a destination PE out of several connected neighbors. Using the companion channels, to specify either the vertical or the horizontal channel requires only one configuration bit.

Each PE in our architecture is a simple programmable predicated ALU similar to the PE defined in ADRES with the goal of accelerating the program kernel. ADRES assumed that in a single hop (a single cycle) a signal can travel across 5 PEs [11]. RCP assumed that one hop can travel across 3 PEs because of the complexity of its crossbar-like *connection*

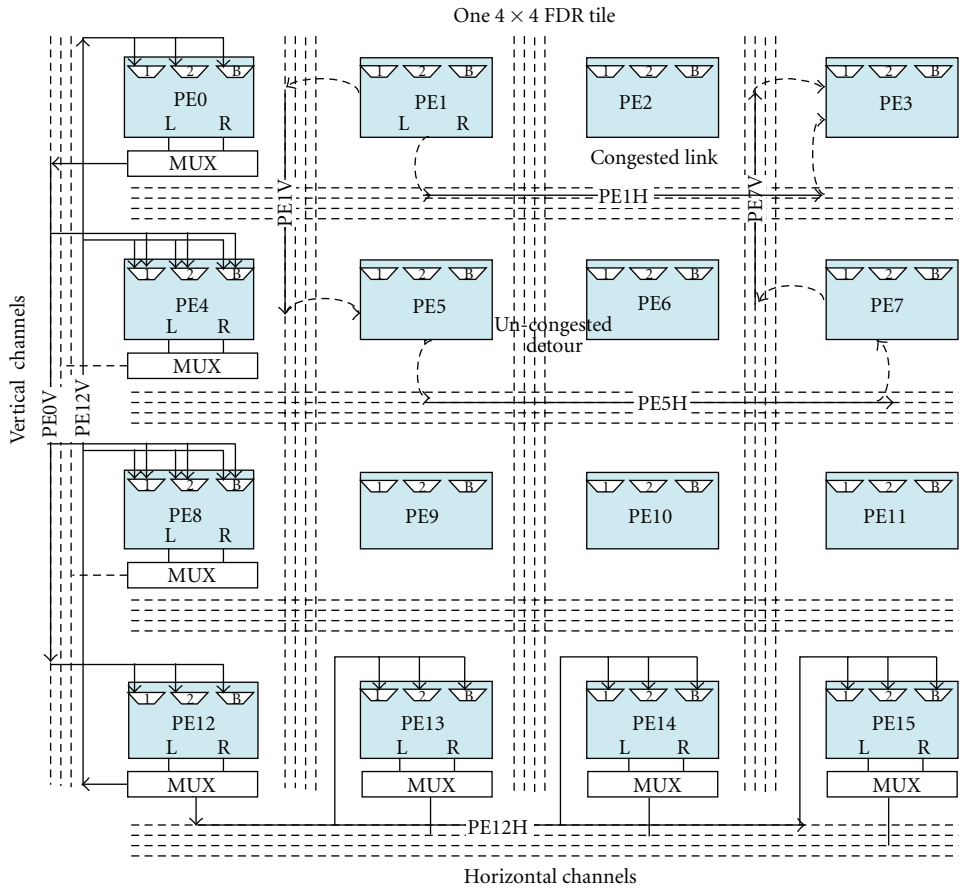



Figure 3: Companion channels and fast data relay.

*box* design [12]. RAW used complex MIPS core as PE, thus one hop can only travel across 1 PE [13]. Given the similarity of our enhanced PE to PE used in ADRES, we assume that one hop of data transmission can travel across 4 PEs. As a result, our wire-based companion channels enable a corner-to-corner transmission in a 4 × 4-tiled FDR architecture to finish in 2 cycles (PE0 → PE12 → PE15) via PE0 vertical channel (PE0V) and PE12 horizontal channel (PE12H) as shown in Figure 3.

It should be noted that ADRES connects PEs with dedicated point-to-point links without the capability of multi-cycle routing and buffering as proposed in our FDR-CGRA. The bypassing registers and the companion channels enable us to do data communication in a more flexible way. The right upper corner of Figure 3 illustrates a possible scenario to do multi-cycle communication if data needs to be sent from PE1 to PE3 and the compiler finds the horizontal channel PE1H connecting from PE1 to PE3 is congested, then it can decide to use a multi-cycle detour through PE5 and PE7 by utilizing their bypassing registers and companion channels PE1V, PE5H, and PE7H to avoid the congestion. Comparing with RCP, FDR-CGRA supports 2D communication, rather than the 1D ring-shape communication in RCP. Furthermore, the companion channel also allows shared data to be sent concurrently to multiple PEs in the same row/column. These capabilities help alleviate the data route congestion problem experienced in other works (e.g., [15]).

Each communication link used in FDR involves three components: sender PE, companion channel, and receiver PE. We call such a communication path *inter-PE link*. It is desirable to reduce the total amount of inter-PE communication to reduce the latency and power. Next, we introduce the concept of non-transient copy to mitigate the inter-PE link usage in our solution.

*2.4. Non-transient Copy.* All PEs on the multi-cycle communication path except the sender are called *linking PEs*. During the multi-cycle FDR, the source scalar data will be transmitted to a receiver PE through linking PEs. Along the transmission, all linking PEs can possess a local copy of the origin data. Because our PEs contain bypassing register files, multiple local copies from different source PEs can coexist in the bypassing register file simultaneously. Furthermore, we can classify these local copies into two categories: *transient copy* and *non-transient copy*. Here we use "transient" to refer to a software property of a value hold in registers. We call a value a transient copy if the register holding the value will be recycled immediately after the value being fetched by the first trailing-dependent operation. In contrast, a non-transient

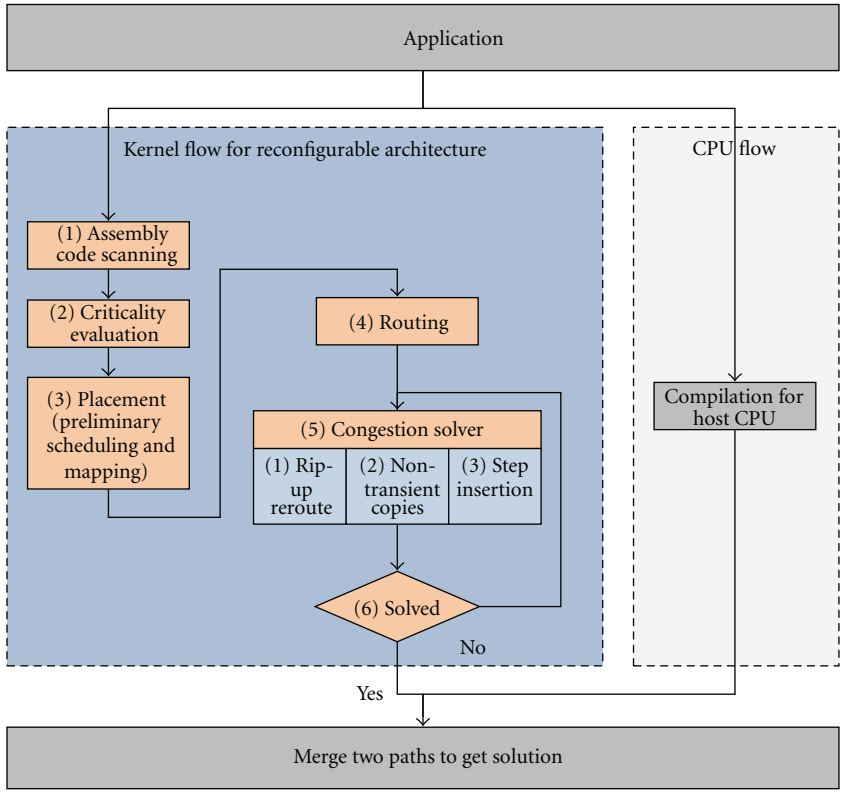



Figure 4: Overall flow.

copy will stay in the register until no more operations depend on it. Note that physically there is no difference between transient copy and non-transient copy since they both reside in the bypassing register file. Conceptually, the compiler treats them differently at compile time according to how they will be utilized. Since ADRES and RCP do not have bypassing register files to hold multiple copies, they need to recycle registers as soon as possible.

As will be shown in our experimental results, using non-transient copies can significantly benefit applications where some source data need to be reused by multiple dependent operations. Because it in effect distributes a source operand to several PEs and keeps them alive as non-transient copies, a dependent operation can probably find a local copy in its vicinity, instead of getting it from the origin PE. This can decrease the number of inter-PE links used.

From now on, we call those linking PEs that can provide local copies *identical start points*. Whether to keep the scalar data in linking PEs bypassing register files as a transient copy or a non-transient copy is decided at the routing stage (Section 3.2).

## 3. Supporting Compiler Techniques

Because FDR-CGRA is intended to accelerate the kernel code of an application, the overall design flow shown in Figure 4 can branch into two separate paths. On the right, the non-kernel code is compiled in a traditional manner targeting the host CPU. On the left is the flow to map kernel code operations onto the FDR-CGRA, which is the focus of our study.

First, we choose *low level virtual machine* (LLVM) [23] to compile the computation-intensive kernel code in C into assembly code. LLVM has its own architecture-independent virtual instruction set, and it can extensively rename registers during compilation, which is essential to avoid hitting the unnecessary IPC wall due to the limited number of registers that can be used by a compiler targeting specific processor architecture [24]. The assembly code of the kernel function generated by LLVM with the optimization option "–O3" is then scanned by a parser to identify data dependencies, and a directed acyclic graph (DAG) representing the data dependency among operations in the kernel code is constructed in step 1 of Figure 4. Step 2 will evaluate the criticality of each node in the DAG. This will be discussed in detail in Section 3.1.

Later on, steps 3–6 are used to find a valid operation mapping honoring resource constraints. Step 3 provides a preliminary placement for operations. Steps 4 and 5 refine it by doing routing and solving congestions. The problem of mapping kernel code onto FDR-CGRA can be formulated as follows: given a DAG representing the data dependency among operations, find a valid scheduling, mapping, and data communication scheme of all operations on the *routing region* so that all data dependency is satisfied and no resource conflict exists, where the routing region is constructed as shown in Figure 5 by the following steps: (1) line up all PEs in the two-dimensional computation grid along the $x$

axis. The PEs on the grid are marked as PE($m, n$), where $m$ is the PE ID and $n$ is this PE schedule step. (2) Duplicate the PE row $L$ times along the time axis (the $y$ axis) to form a routing region, where $L$ is achieved through a list scheduling algorithm honoring the resource constraint. (3) Add edges between any nearby PE rows to reflect the available connectivity among PEs according to the connectivity specification in the architecture, that is, companion channels specified in Section 2.3. Figure 5(b) shows a sample routing region for a small group of PEs specified in Figure 5(a). The solid lines indicate inter-PE connections and the dash lines indicate intra-PE communication. Such a routing region incorporates all the feasible connections between PEs along the scheduling steps and provides a working space to carry out the compilation tasks.

Figure 5(c) is a DAG representing data dependency among operations. In this DAG, (1) solid edges represent single-cycle data dependency and (2) the dashed edge represents multi-cycle dependency, which requires FDR. Our compilation process consists of two phases: *placement frontend* and *routing backend*. The placement frontend produces an operation mapping solution in the routing region and leaves the detailed resource conflict problems to be resolved in the routing backend. This is similar to the physical design stages where a placer determines the location of cells and the routing determines the detailed wiring of the cells. In our case, each cell would be an operation. Figure 5(d) shows a placement and routing result for the DAG in Figure 5(c). In cycle 1, nodes 1 to 4 are mapped to PEs 0 to 3, respectively. In cycle 2, node 5 is mapped to PE 1 and node 6 to PE 3. Note that the output of node 2 is kept in PE 1 bypassing register file, which is then used by node 8 in cycle 3.

We would like to emphasize that explicitly using a dedicated routing phase to solve congestion can enable us to achieve higher ILP performance than modulo scheduling used in CGRA compilers from [11, 12, 14]. When resource conflict due to the competition for the same resource by consecutive iterations in a loop cannot be resolved, modulo scheduling algorithm resorts to increasing the *iteration interval* (II), which has negative impact on ILP performance. Meanwhile, although modulo scheduling is commonly used to optimize loops with arbitrary large number of iterations, the performance could suffer from the unsaturation of operations during the wind-up and wind-down phases. This problem is particularly severe for a compact loop body with small number of iterations. Unfortunately, many applications demonstrate this property because of the popularity of partitioning large kernels into small pieces for data localization. In contrast, our two-phase procedure can virtually discard the II constraints by unrolling the whole loop and explicitly resolving resource conflicts during the backend data routing stage. Considering that the routing algorithm we borrowed from the physical design automation field can typically handle hundreds of thousands of instances, this dedicated routing phase enables us to work on the fully unrolled kernel loops efficiently.

*3.1. Placement Frontend for FDR .* A criticality evaluation needs to be done first for all nodes in the DAG to guide placement. Note that we will use the term node and operation interchangeably from now on. The node criticality is evaluated as *slack* defined as

$$slack(op) = ALAP\_level(op) - ASAP\_level(op).$$

*ALAP_level* (*ASAP_level*) is the as-late-as-possible scheduling level (as-soon-as-possible scheduling level) for an operation computed by the ALAP (ASAP) scheduling algorithm [25]. If the slack is zero, there is no margin to delay the execution of the operation. If the slack is a positive value, the operation can be delayed accordingly without hurting the overall schedule length.

The placement frontend carries out operation scheduling and operation-to-PE mapping. We perform a list scheduling [25] guided by operation criticality such that critical operations will be placed on PEs that are directly connected with companion channels and noncritical operations will be placed on PEs without direct companion connecting them.

Algorithm 1 outlines the algorithm. First of all, every unprocessed node (operation) whose dependency is satisfied (i.e., its predecessor nodes, including PI nodes, are already scheduled and mapped) in the DAG is collected into *ReadyNodeSet*. Given the *ReadyNodeSet*, *FreePESet* (available PEs) and the current schedule step $S$ as inputs, the algorithm maps the nodes from *ReadyNodeSet* to PEs in *FreePESet* and output a *MappedNodeSet*. Two constraints are considered here (lines 2–12 of Algorithm 1).

*3.1.1. Functionality Constraint.* This is to make sure that the candidate PE has the ability to perform the required computations because the PEs could be heterogeneous. For example, not all ALUs can perform memory load/store operations as explained in Section 2.1. We use *CapablePESet(n)* to identify the set of PEs that are capable of performing the operation $n$ (line 3).

*3.1.2. Routability Constraint.* Although detailed routing scheme is determined at routing backend, a hint of routability is still used to help Algorithm 1 to determine mapping of nodes by calling *RoutableSet(p, l)* (line 8-9). Recall that FDR may need multiple steps to send data from a source PE to a destination PE. Given a PE $p$, the $l$-step reachable set of $p$ is defined as all the PEs which can be reached within at most $l$ hops starting from $p$. Then the candidate PEs are filtered out by taking the intersection of candidate sets generated according to functionality constraints, routability constraints, and availability constraints (line 10).

As an example, assume operation node $b$ has two source operands: one is on PE1, and other is on PE2. Assume we have

$$CapablePESet(b) = \{\text{PE0 PE2 PE4 PE8}\},$$
$$RoutableSet(PE1, l) = \{\text{PE4 PE5 PE7}\},$$
$$RoutableSet(PE2, l) = \{\text{PE3 PE4 PE9}\}.$$

The intersection of the above three sets is PE4, indicating that PE4 is the only mapping candidate for node $b$. If PE4 turns

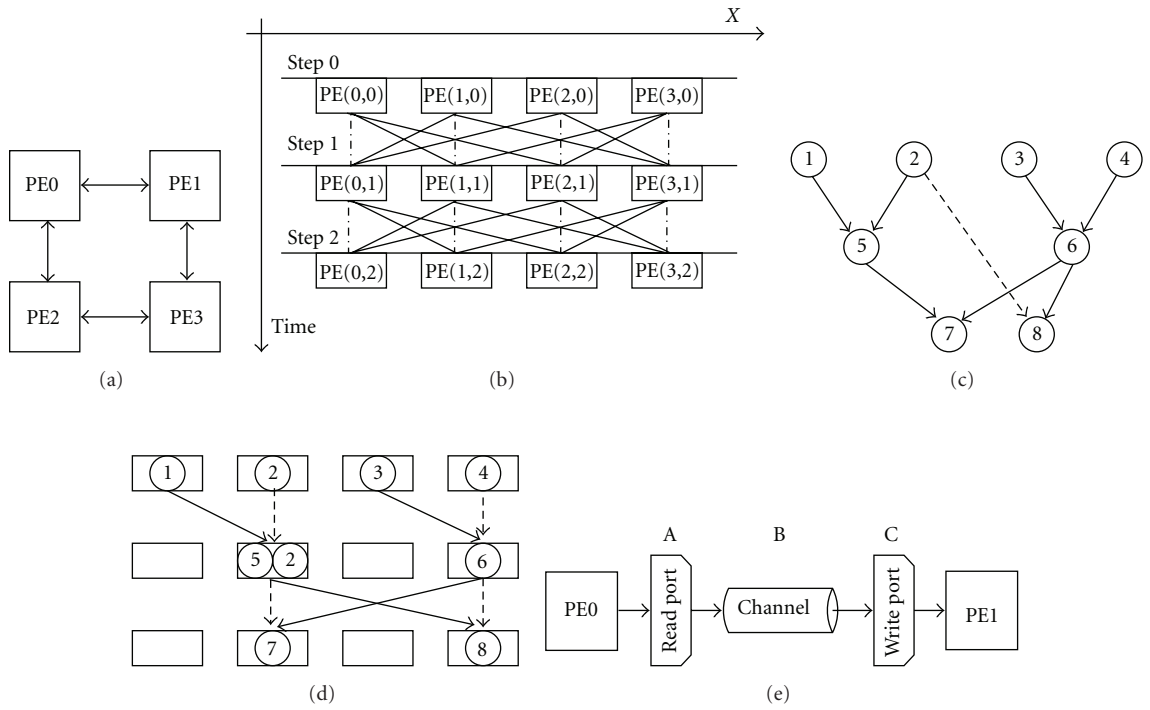



Figure 5: (a) A $2 \times 2$ PE array, (b) the routing region, (c) the DAG, (d) the mapping and routing solution, and (e) constraint model

**Input:** *ReadyNodeSet*, *FreePESet*, current schedule step *S*
**Output:** *MappedNodeSet*
(1) **for** *each* $PE_i$ *in FreePESet* {CandidateNodeSet($PE_i$) = {empty};}
(2) **for** *each* node *n in* ReadyNodeSet {
(3) CapablePESet(*n*) = all PE capable of the function of *n*;
(4) *S_src1* = the time when *n*'s operand 1 is produced;
(5) *L_src1* = the PE where *n*'s operand 1 is stored;
(6) *S_src2* = the time when *n*'s operand 2 is produced;
(7) *L_src2* = the PE where *n*'s operand 2 is stored;
//all PEs which can be reached within
//(*S-S_src1*) cycles from PE *L_src1*;
(8) *RoutableSet1* = RoutableSet(*L_src1*, (*S-S_src1*));
//all PEs which can be reached within
//(*S-S_src2*) cycles from PE *L_src2*;
(9) *RoutableSet2* = RoutableSet(*L_src2*, (*S-S_src2*));
(10) *CandidatePESet* = *RoutableSet1* ∩*RoutableSet2* ∩*FreePESet* ∩*CapablePESet*(*n*);
(11) **for** *each* $PE_i$ *in CandidatePESet*{Add *n* to *CandidateNodeSet* ($PE_i$);}
(12) }
(13) *MappedNodeSet* = {empty};
(14) **for** *each* $PE_i$ *in CandidatePESet*{
(15) Sort all nodes in *CandidateNodeSet* ($PE_i$) according to criticality;
(16) *M* = most critical from *CandidateNodeSet* and $M \notin$ *MappedNodeSet*;
(17) Place *M* on $PE_i$ at schedule step *S*;
(18) Insert *M* into *MappedNodeSet*;
(19) Remove *M* from *ReadyNodeSet*;
(20) }

Algorithm 1: Placement frontend (simultaneous scheduling and mapping).

out to be already occupied (PE4 not free), node $b$ cannot be mapped in this schedule step and will fall to later schedule steps.

If there are more than one candidate nodes to map into a PE as indicated in *CandidateNodeSet*, the node with higher criticality will have a privilege over those with lower criticality for mapping (line 13–19). Those successfully mapped nodes will be moved to the *MappedNodeSet* (line 18), while the unmapped nodes will still remain in the *ReadyNodeSet* to be processed in scheduling steps thereafter. This procedure will continue until all nodes are processed.

*3.2. Routing Backend for FDR.* The routing backend is used to solve resource conflict in detail. We use an algorithm inspired by global maze routing [26] to find inter-PE communication schemes that can efficiently utilize bypassing registers introduced in Section 2.2. Our algorithm also determines which operands stay in bypassing registers as non-transient copies and which are not.

Each operand used in the DAG is identified with its own ID and its associated location in the routing region in the format of PE$[m, n]$, where $m$ stands for the ID of PE that hosts the operand and $n$ stands for the schedule step in which the operand is ready for R/W (read/write). The inputs to the routing procedure are a list of routing requirements represented as ($src \rightarrow dest$) pairs, where *src* and *dest* are in the above format. Figure 6 shows an example. Assume three operations op1, op2, and op3 all need an operand $M$ from PE $[1, 1]$ as indicated in Figure 6(a) with 3 routing requirements:

$$\text{op1: } M, \text{PE}[1,1] \rightarrow \text{PE}[2,4],$$
$$\text{op2: } M, \text{PE}[1,1] \rightarrow \text{PE}[0,4],$$
$$\text{op3: } M, \text{PE}[1,1] \rightarrow \text{PE}[3,5].$$

*3.2.1. Solve Routing Congestion with Rip-Up and Reroute.* An initial routing solution usually has congestion. We improve the quality of routing solutions iteratively. A commonly used way to solve congestion is by rip-up and reroute [26] as shown in Algorithm 2: (1) pick a routed path (line 4); (2) rip-up (or break) the path by deallocating all the resources (i.e., channels and ports) taken by this path except the source PE and the destination PE (line 8); (3) based on the existing congestion information of the routing region, find an uncongested detour and reroute the path (line 11–14). These steps are iterated for all paths until the congestion is eliminated or it stops when no improvement can be achieved. After rip-up and reroute, a routing solution for the previous problem is shown in Figure 6(b), in which four inter-PE links (solid arrows) are used.

*3.2.2. Detail Route for Each Inter-PE Communication Path.* To detail route for each communication path, a maze routing procedure "route_path" in line 11 of Algorithm 2 is called to find routing solution for each requirement entry. Note that a set of special data structures used for CGRA is used to provide routing guidance for "route_path." First, we apply *channel constraints*, which are channel capacities on vertical and horizontal companion channels and apply *port constraint* which is an upper bound on the number of total simultaneous accesses to the read/write ports of bypassing register files in PEs to reflect the bandwidth constraint of a physical register file (Figure 2). Second, each inter-PE communication link is modeled as a cost function of three parts (Figure 5(e)): read port, channel, and write port. Whenever an inter-PE link is used, the capacity of the affected channel and ports will decrease and the cost of using this link will increase proportional to the congestion on it. Third, each PE has a PE score recording the cost of the cheapest path (with least congestion) from the source PE.

At the beginning of "route_path," a queue contains only the source PE. Starting from the source PE by popping it up from the queue, multiple-directional scores are calculated for its trailing PEs, that is, directly connected neighboring PEs. PE scores for those trailing PEs may be updated to record the changes for the cheapest paths leading to them. Any PE whose PE score is updated will be pushed into the queue for further exploration. The process is quite similar to wave front propagation, and the router stops when the queue is empty. After wavefront propagation is done by "route_path," a "backtrace" procedure (line 13) starts from the end point and backtraces along the cheapest direction at each backtrace step in the routing region "cost_grid" until reaching the start point. This backtraced path "new_p" is reported as the cheapest path.

*3.2.3. Utilize Non-Transient Copies.* What differentiate our algorithm from ordinary routing algorithm is that we integrate the utilization of features of non-transient copy in the routing algorithm.

In the example of Figure 6(b), after the first route from $M$ on PE$[1, 1]$ to op1 on PE$[2, 4]$ is established via the path PE$[1, 1] \rightarrow$ PE$[3, 2] \rightarrow$ PE$[2, 3] \rightarrow$ op1. Both PE2 and PE3 have once possessed a copy of $M$ during this procedure. In fact, the copy in PE3 can be reused by op3 later on at step 5, as long as the register holding value $M$ in PE$[3, 2]$ will not be reclaimed immediately after PE$[2, 3]$ fetches it. This is achieved by marking the copy of $M$ in PE$[3, 2]$ as a non-transient copy. In contrast, for the copy of $M$ in PE$[2, 3]$, except op1 no other operation needs it. Therefore register holding $M$ in PE2 can be marked as transient and reclaimed immediately after op1 consumes it to save register usage. To decide whether a copy should be transient or non-transient, a special data structure "regDuplicatorLst" is used in Algorithm 2 to help making the decision. "regDuplicatorLst" is implemented as a lookup table storing the identical start points for each origin source operand. The index to the table is the origin source operand, and the contents pointed by the index are all identical start points organized as a list. This lookup table is updated (line 15) every time after the cheapest path is found by the backtrace procedure by inserting all new identical start points into the entry of their associated origin source operands. An example of a "regDuplicatorLst" entry is shown below. After the first route to op1 in Figure 6(b) is found by "backtrace," the origin source $M$ may be found in three different places:

$$M\text{: PE}[1,1], \text{PE}[3,2], \text{PE}[2,3].$$

```
Input: &routingPathLst // routingPathLst stores routing requirements
Output: a new less congested routing solution stored in routingPathLst
1     for (i = 0; i < iMax; i++) {
2     regDuplicatorLst.clear(); // clear regDuplicatorLst before new iteration
3     newRoutingPathLst.clear();
4     for each path p in routingPathLst {
5      src = p.origin;    // get start_point of p
6      dest = p.destination;    // get end_point of p
7      //step 1: ripup
8      ripup_path(p);        // ripup p by release all resources used by p
9      end_step = getReadyStep(dest);
10       //step 2: reroute
11       cost_grid = route_path(src, end_step, &regDuplicatorLst);
12       //step 3: backtrace
13       new_p = backtrace(dest, &src, cost_grid);
14       add(new_p, newRoutingPathLst);
15       update(regDuplicatorLst, new_p);
16 }
17       routingPathLst = newRoutingPathLst;
18 }
```

Algorithm 2: ripup_reroute.

As we pointed out before, an identical start point can serve as an alternative source for the same operand. To take this into consideration, procedure "route_path" of Algorithm 2 is modified accordingly to push all identical start points for the origin source into the queue at the beginning of the routing. As a result, the wavefront propagation can actually start from multiple sources. But only one source that produces the cheapest route found by "backtrace" will be used as actual start point. If this actual start point is different from the origin source, it means this data relay route actually reuses an operand that is deposited by another route before. To allow this reusing, this start point is marked as a non-transient copy.

In the simple example of Figure 6(c), by maintaining the identical start points in "regDuplicatorLst," the backtrace for op3 is able to find the cheapest route solution, which is to reuse the value $M$ once stored in the bypassing register, for example, BR[2], of PE[3, 2]. Then BR[2] is declared as nontransient copy thus can be kept untainted in PE[3, 2] for future reuse by op3 without establishing a physical inter-PE link from the origin of $M$ in PE[1, 1]. With non-transient copies, the solution can make less use of inter-PE links, for example, in Figure 6(c) only three inter-PE links are used while four are used in Figure 6(b). Moreover, during the routing for op2, with the "regDuplicatorLst," Algorithm 2 will find it can get a copy of $M$ from either PE[1, 3] or PE[2, 3] in Figure 6(c). With more choices, the router in Algorithm 2 can effectively select the least crowded communication scheme to avoid congestion.

Non-transient copies usually have longer life cycles than transient copies and serve as data origins for multiple trailing-dependent operations. Our compiler can trace the last access to a non-transient copy at the compile time and decide to reclaim this non-transient copy after this last usage.

*3.2.4. Schedule Step Relaxation.* In some cases, applying the above techniques still cannot resolve all the congestions, and *schedule step relaxation* (SSR) will then be invoked as the last resort. SSR utilizes a well-known fact that routing sparsely distributed nodes on a larger routing region will result in less congestion. SSR in effect increases the size of the routing region along the time axis by inserting extra steps into the most congested regions. Trading area for routability guarantees that congestion will be reduced to zero. As the last resort, a greedy algorithm is used to do SSR as follows: starting from the most congested step, insert an extra *relaxation step*, then redo rip-up and reroute on the extended routing region, and continue doing so until all congestions are solved. The side effect of SSR is the increased schedule length.

## 4. Experimental Results

Experiments are performed on an architecture configuration shown in Figure 1. The reconfigurable array consists of 4 tiles. Each tile has 16 PEs organized in a $4 \times 4$ mesh. The PEs in a single tile communicate with each other as specified in Section 2.3. The data relay ports on the bypassing register for read and write are bounded by 2, respectively, to reflect the physical register file bandwidth constraint. Similarly, load/store operations are limited to be performed only by PEs on the diagonal line to reflect memory bandwidth constraint. Other computational operations can be performed by any PEs. The latency of each operation is set to 1. LR has 16 entries. To explore the best performance that can be achieved using BR, the size of BR is not fixed. However, as the results shown in Section 4.3, a fairly small amount of BR is used, thus practical. The architecture in Figure 1 can be configured in two ways: (1) *large Cfg.*, all 64 PEs can be used to map

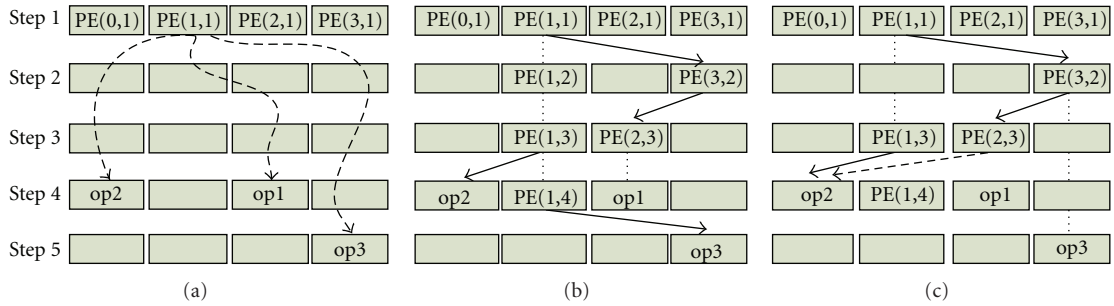



Figure 6: (a) Initial routing requirements, (b) data relay solution, and (c) solution of data relay with register duplication.

a benchmark program, (2) *small Cfg.*, a benchmark program is limited to be run on a single tile, but four independent programs can run concurrently on four tiles. Each Cfg. has its own merit to be explained in Section 4.2.

Benchmarks are extracted from two open source applications: xvid 1.1.3 and JM 7.5b. xvid [27] is an open source package consisting of various codecs for various video formats. JM v7.5b [28] is a reference implementation of the H.264 video codec. Five subroutines are extracted from the xvid package: idct_row, idct_col, interpolation8 × 8_avg4_c, interpolate8 × 8-_halfpel_*hv*_c, and SAD16_c. IDCT is a fixed-point inverse discrete cosine transformation working on rows and columns of the 8 × 8 pixel blocks. This is the same as that used in [11] and [12]. Interpolation8 × 8_avg4_c and interpolate8 × 8_halfpel_*hv*_c are subroutines to calculate pixel values for 8 × 8 blocks by interpolating pixels from neighbor image frames. SAD16_c is used in motion estimation to calculate the similarity of two 16 × 16 pixel blocks. Another four subroutines are extracted from the H.264 reference implementation JM v7.5b: getblock(V), getblock(H), getblock(HV), and getblock(VH). They calculate the pixel values for a 4 × 4 block by interpolating from neighbor image frames.

We should point out that the number of loop iterations in media applications is bounded by fine-grained partitioned data set, usually for better image quality. For instance, each H.264 loop typically operates on a 4 × 4 or 8 × 8 pixel block. This type of usage gives rise to the opportunity to exploit ILP within loops by fully unrolling, which fits perfectly to FDR-CGRA typical usage.

*4.1. Execution Trace.* Our compiler implemented in C++ can dump a placed and routed execution trace for the kernel code. Then a simulator implemented in PERL is used to validate the result. We use PERL because it supports text array very well, and text arrays are used to represent instruction, channel, and port configurations in our code. The simulator not only simulates the data flow on FDR-CGRA but also verifies on the fly for the satisfaction of three sets of resource constraints: channel constrains, port constraints and resource constraints. Table 4 shows an abbreviated execution trace for idct_row scheduled on an architecture depicted in Figure 1. To save space, only operations on one tile are shown and the other three tiles are omitted. 16 PEs in the tile are numbered from 0 to 15. Operations are represented as parenthesized letters. Round bracket ones are of one unrolled iteration (ite1), while square bracket ones are of another iteration (ite2). Operations from the same iteration can be scheduled across tiles when necessary. Both iterations start at the first cycle. In our approach, iterations compete for PE resource and also complement each other when the other is waiting for data: for example in Table 4, nine ops are from ite1 in cycle 5, while six ops are from ite2, while in cycle 6, five ops are from ite1 and ten ops are from ite2. In addition, we can see that the result respects the resource constraints by (1) only allowing PE0, 5, 10, 15 to perform load/store operations ($L$) due to memory bandwidth constraint (2) and only allowing at most 2 reads and 2 writes to each PE bypassing register file due to the port constraint (2R2W) we set. In Table 4, the "write (read) requests" column shows, for each PE, how many write (read) requests are served at each cycle. Each single digit from left to right in a row corresponds to the number of write (read) requests on PE0 to PE15, respectively. Each column of these digits reflects the write (read) requests for its corresponding PE over time. The "Communication Vol.," where each * represents an inter-PE link usage, shows the amount of inter-PE link usage at each cycle. It is clear that computation and communication are indeed concurrently executed every cycle due to dedication of bypassing register, except cycle 1, 7, 10, 12, where extra relaxation steps are used to solve routing congestion.

*4.2. Performance Analysis.* Modulo scheduling-based code scheduling approaches [11, 15] have advantage to handle loops with a large number of iterations because they can reuse the scheduled code on PEs throughput the iterations. However, the inflexibility of modulo scheduling may limit its performance for a class of applications that have a special communication pattern. This type of applications, represented by video applications, usually "broadcast" shared data from one PE to multiple receiving PEs. As reported in [15], an edge centric modulo scheduling does not perform well for H.264 if an CGRA does not provide complex routing schemes including using diagonal channels.

The performance of benchmark programs on FDR-CGRA, as well as on ADRES and RCP, is shown in Tables 1 and 2. The benchmark programs are mapped on

Table 1: Large configuration results.

| Arch. | App. | Large Cfg.: 4 tiles, each tile has 4 × 4 PEs | | | | |
|---|---|---|---|---|---|---|
| | | Ops | Cycles | Avg. IPC | Perf. gain | Efficiency |
| ADRES | idct_row(8 × 8) | — | — | 27.7 | — | 43% |
| FDR-CGRA | idct_row(8 × 8) | 857 | 24 | 35.7 | 29% | 56% |
| ADRES | idct_col(8 × 8) | — | — | 33.0 | — | 52% |
| FDR-CGRA | idct_col(8 × 8) | 1185 | 33 | 35.9 | 9% | 56% |
| FDR-CGRA | Interpolate 8 × 8_avg4_c | 1193 | 40 | 29.8 | — | 47% |
| FDR-CGRA | Interpolate 8 × 8_halfpel_*hv*_c | 1295 | 38 | 34.1 | — | 53% |
| FDR-CGRA | sad16_c(16 × 16) | 3441 | 106 | 32.5 | — | 51% |

Table 2: Small configuration results.

| Arch. | App. | Small Cfg.: 1 tile, 4 × 4 PEs each tile | | | | |
|---|---|---|---|---|---|---|
| | | Ops | Cycles | Avg. IPC | Perf. Gain | Efficiency |
| RCP | idct(row+col) | — | — | 9.2 | — | 57% |
| FDR-CGRA | idct(row+col) | 2042 | 184 | 11.1 | 21% | 69% |
| FDR-CGRA | interpolate8 × 8_avg4_c | 1193 | 136 | 8.8 | — | 55% |
| FDR-CGRA | interpolate8 × 8_halfpel_*hv*_c | 1295 | 135 | 9.6 | — | 60% |
| FDR-CGRA | sad16_c(16 × 16) | 3441 | 339 | 10.2 | — | 63% |
| ADRES | get_blocks (64 PEs) | — | — | 29.9(64 PEs) | — | 47% |
| FDR-CGRA | get_block(H) | 340 | 38 | 8.9 | — | 56% |
| FDR-CGRA | get_block(V) | 296 | 37 | 8.0 | — | 50% |
| FDR-CGRA | get_block(V+H) | 899 | 93 | 9.7 | — | 60% |
| FDR-CGRA | get_block(H+V) | 900 | 95 | 9.5 | — | 59% |
| FDR-CGRA | Adjusted Avg. (4 tiles) | — | — | 36.1(4 tiles) | 21% | 56% |

both a large and a small Cfg's. In the large Cfg., our approach achieves an average IPC of 35.7/35.9 for idct_row/idct_col, outperforming 8 × 8-FU ADRES architecture reported in [11] by 29% and 9%, respectively. In the small Cfg., we compare our results with 16-issue RCP reported in [12]. Our result of IDCT scheduled on a single tile (16 PEs) outperforms the 16-issue RCP IPC = 9.2 by 21%. As to the get_blocks, we observed that some modes—get_block(H) and get_block(V)—have significant fewer ops than other benchmark programs. Spreading a small number of ops on a large number of PEs would artificially introduce many multi-hop routing requirements. Given that each get_block can indeed work independently of others, a more efficient way is to schedule each get_block on a single tile and allow four instances of different get_block's to run concurrently. With this 4x adjustment to the get_block's average IPC in Table 2, the final average IPC for get_block is larger than 36, which is better than the "in-house" optimized results (avg. IPC = 29.9 for ADRES) in [22]. Note that the IPC values for ADRES and RCP are directly quoted from their published results. No data is reported for the interpolation and SAD benchmarks from either ADRES or RCP.

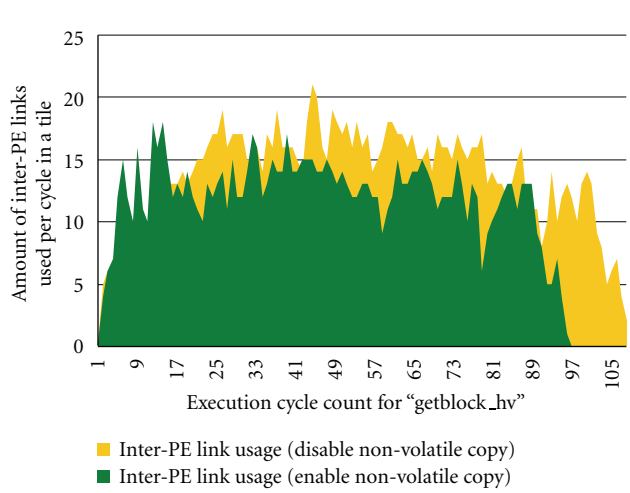
In the rightmost column of Tables 1 and 2, efficiency of a schedule is defined as the quotient of average IPC divided by total number of PEs. It can be observed that for the same application the large Cfg. has the smallest latency (under the *cycles* column) because it can use more resources, while the small Cfg. always has higher efficiency. This is because scattering operations on large Cfg. may introduce more operation "bubbles", where no useful work is actually done by a NOP operation for dependency or communication reasons. This fact suggests that the large Cfg. can be used when reducing execution latency is the sole goal, while the small Cfg. is suitable for applications where efficiency and performance are both important.

We should point out that one important reason that the proposed FDR-CGRA outperforms ADRES is the use of companion channels and non-transient copies. When necessary, the wire-based companion channel naturally allows "broadcast" of a datum to multiple PEs in the same row or column. At the same time, non-transient copies not only allow reusing of datum that needs to be shared by multiple PEs but also increase the path choices during routing.

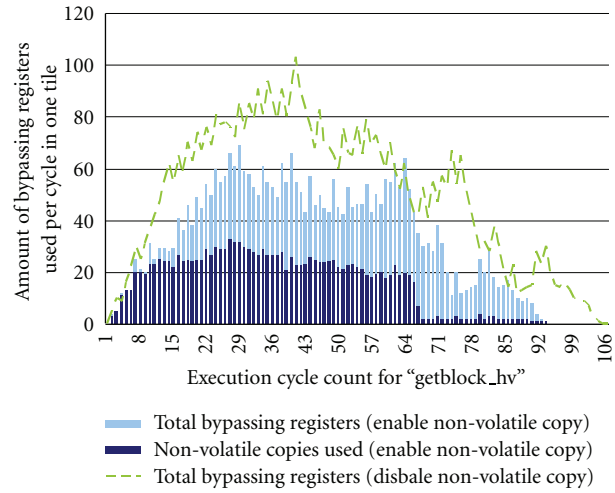
The comparison with RCP is also interesting because RCP also allows concurrent data routing and computations. However, it is limited to one dimension. The experimental results demonstrate that our proposed compiling technique effectively exploits the performance potential of FDR 2D concurrent data routing and computations.

Table 3: Bypassing register usage profile and effects of using non-transient copy on a single tile (16 PE) configuration.

| | Bypassing register usage profile: per PE per cycle | | | | | Performance impact | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Average | | Peak | | % Nontransient | Amount of inter-PE links | | | IPC | | |
| | Disable | Enable | Disable | Enable | Enable | Disable | Enable | Delta | Disable | Enable | Delta |
| idct(row+col) | 2.8 | 2.3 | 19 | 20 | 45% | 2357 | 2245 | −5% | 10.9 | 11.1 | 2% |
| interpolate8 × 8_avg4_c | 4.8 | 3.1 | 21 | 14 | 58% | 1707 | 1333 | −22% | 6.4 | 8.8 | 38% |
| interpolate8 × 8_halfpel_*hv*_c | 3.7 | 3.9 | 19 | 16 | 45% | 1835 | 1621 | −12% | 8.3 | 9.6 | 16% |
| sad16_c(16 × 16) | 1.0 | 0.8 | 6 | 5 | 54% | 4752 | 4000 | −16% | 9.8 | 10.2 | 4% |
| get_block(horizontal) | 1.0 | 0.9 | 6 | 5 | 22% | 455 | 438 | −4% | 8.1 | 9.0 | 10% |
| get_block(vertical) | 1.6 | 1.0 | 8 | 5 | 52% | 513 | 400 | −22% | 5.7 | 8.0 | 41% |
| get_block(V+H) | 4.2 | 2.2 | 18 | 10 | 45% | 1469 | 1150 | −22% | 7.9 | 9.7 | 23% |
| get_block(H+V) | 3.1 | 2.3 | 13 | 8 | 43% | 1455 | 1148 | −21% | 8.4 | 9.5 | 13% |
| Average | | | | | | | | −15% | | | 18% |




(a) Inter-PE link usage for each execution cycle with nontransient copy enabled and disabled




(b) Bypassing register usage for each execution cycle with nontransient copy usage breakdown

Figure 7: Performance impact of nontransient copies.

*4.3. Effect of Non-Transient Copies.* To test the effect of using non-transient copies, we rerun all the applications in Table 2 without using non-transient copies for a single tile configuration. The experimental results in Table 3 indicate that our algorithm produces solutions that only use an affordable amount of bypassing registers. On average, each PE only uses several bypassing registers at each cycle (column 2-3) and the peak bypassing register usage of a single PE ranges from 5 to 21 depending on applications. When non-transient copy usage is enabled, 22–58% of the total bypassing registers hold non-transient copies.

Table 3 also shows the performance impact of enabling non-transient copy. We can view its effects from three perspectives: (1) the primary goal of using it is to mitigate data relay congestion by allowing data reusing. With less congestion, the applications can finish sooner. (2) Distributing non-transient copies across a tile enables a PE to find a local copy in its vicinity and thus can reduce the usage of resource-and-power-consuming inter-PE links. (3) With less usage of inter-PE links, the total number of bypassing registers required can actually be reduced, although those non-transient copies do not recycle as fast as transient copies. These three effects are reflected in Figure 7 for application "getblock_hv." The usage for inter-PE links and usage for bypassing registers are plotted for every cycle. Obviously in Figure 7(a), non-transient copies enable the application to finish earlier and use less inter-PE links in total. It can be seen in Figure 7(b), with non-transient copy enabled, the proportion of non-transient copies to total bypassing registers is close to 1 in the first few cycles ranging from cycle 1 to 12 that correspond to loading input values from a reference image block. It means most of these loaded input values will be reused later on. Correspondingly, in Figure 7(a) the inter-PE link usage is also higher in the first a few cycles because non-transient copies need to be distributed across the tile as soon as possible for further reusing. But interestingly the total amount of bypassing registers used can still be kept lower with non-transient copy enabled. This is because that by reusing non-transient copies, the compiler does not need to allocate as many fresh

Table 4: Execution trace of IDCT (row).

| | PE | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | Communication Vol. | Write requests | Read requests | |
| Cycle 0: | (L) | — | — | — | — | [L] | — | — | — | — | — | — | — | — | — | — | | 0000000000000000 | 1000010000000000 | |
| Cycle 1: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | ∗∗ | 0100100000000000 | 1100100000000000 | |
| Cycle 2: | (P) | (P) | (P) | — | [P] | [P] | [P] | — | — | — | — | — | — | — | — | — | ∗∗∗ | 0011001000000000 | 0111111000000000 | |
| Cycle 3: | (L) | — | — | — | — | (L) | — | — | — | — | (L) | — | — | — | — | (L) | ∗∗∗∗∗∗ | 0000011110100001 | 1000121100200002 | “L”: load |
| Cycle 4: | (C) | (C) | (C) | (C) | (C) | [L] | (C) | (C) | — | — | [L] | (C) | — | — | — | [L] | ∗∗∗∗∗∗∗∗∗∗ | 0111111100110001 | 2222122110210002 | “S”: store |
| Cycle 5: | (A) | (A) | [C] | [C] | [C] | (M) | (M) | (M) | (M) | [C] | [L] | [C] | (M) | — | (M) | (H) | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | 2211221112111011 | 0111201002221001 | “P”: getptr |
| Cycle 6: | (M) | (M) | [A] | [M] | [M] | (A) | [M] | (A) | [C] | [A] | [M] | [M] | (H) | [C] | [C] | — | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | 0011111111111111 | 1101001110111111 | “C”: cast |
| Cycle 7: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | ∗∗∗∗∗∗∗∗∗∗∗∗ | 0121101111101001 | 0110110110100011 | “M”: mul |
| Cycle 8: | (B) | (B) | (B) | (A) | [A] | (M) | [M] | — | [M] | [M] | [H] | — | [H] | — | — | (B) | ∗∗∗∗∗∗∗∗∗ | 1111101000110001 | 1122122111001001 | “A”: add |
| Cycle 9: | (B) | (B) | [B] | (A) | [B] | [M] | (A) | (B) | [A] | [B] | [A] | (A) | — | — | — | — | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | 2112111111020011 | 1101111111111001 | “B”: sub |
| Cycle 10: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | ∗∗∗∗∗∗∗∗∗∗∗∗∗ | 1111111111110100 | 1011001111120111 | “H”: shl |
| Cycle 11: | (B) | (A) | (C) | (B) | [B] | [A] | (A) | (B) | (A) | [B] | (A) | [A] | — | — | — | — | ∗∗∗∗∗∗∗∗∗∗∗∗∗ | 1001011111221001 | 1011110011111000 | “R”: shr |
| Cycle 12: | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | ∗∗∗∗∗∗∗∗∗∗ | 1010110101101101 | 1010211101110000 | |
| Cycle 13: | (M) | (M) | (C) | (C) | [B] | [B] | (C) | [B] | (C) | (C) | [B] | [A] | [A] | [A] | — | [A] | ∗∗∗∗∗∗∗∗∗∗ | 0001100100121111 | 0012110112110111 | |
| Cycle 14: | (A) | (A) | (B) | [C] | [B] | [A] | [C] | [C] | (B) | [C] | (A) | [C] | [C] | — | — | (A) | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | 0111111111210002 | 0102001101122001 | |
| Cycle 15: | (R) | (R) | (C) | (R) | [M] | [M] | (R) | — | (C) | [B] | [A] | — | — | — | — | [B] | ∗∗∗∗∗∗∗∗∗∗∗∗ | 0011101111210002 | 0011001110010001 | |
| Cycle 16: | (C) | (C) | (R) | [C] | [A] | [A] | (C) | (C) | (R) | [C] | [R] | [A] | — | — | — | — | ∗∗∗∗∗∗∗∗ | 0002100100110001 | 2211101100100000 | |
| Cycle 17: | (B) | (B) | (A) | (A) | [R] | [R] | (C) | [R] | (C) | [R] | (S) | [R] | — | — | [C] | (S) | ∗∗∗∗∗∗∗∗∗∗ | 2111001100100011 | 0010002111010010 | |
| Cycle 18: | (C) | (C) | (R) | (R) | [C] | [C] | [C] | [C] | [C] | — | [S] | — | — | — | — | [S] | ∗∗∗∗∗∗∗∗ | 0000011110200011 | 0000221010000010 | |
| Cycle 19: | (R) | (R) | (C) | (C) | [B] | [B] | [A] | [A] | — | — | (S) | — | — | — | — | (S) | ∗∗∗∗∗∗∗∗ | 0000111100200001 | 0011001100000000 | |
| Cycle 20: | (C) | (C) | [R] | [R] | [C] | [C] | — | — | — | — | (S) | — | — | — | — | (S) | ∗∗∗∗ | 0011000000100001 | 0100011100000000 | |
| Cycle 21: | (S) | [R] | [C] | [C] | [R] | (S) | — | — | — | — | [S] | — | — | — | — | [S] | ∗∗∗∗ | 0100010000100001 | 0011100000000000 | |
| Cycle 22: | [C] | [C] | — | — | — | — | — | — | — | — | [S] | — | — | — | — | [S] | ∗∗∗ | 1000000000100001 | 0100100000000000 | |
| Cycle 23: | [S] | — | — | — | — | [S] | — | — | — | — | — | — | — | — | — | — | ∗∗ | 1000010000000000 | 0000000000000000 | |

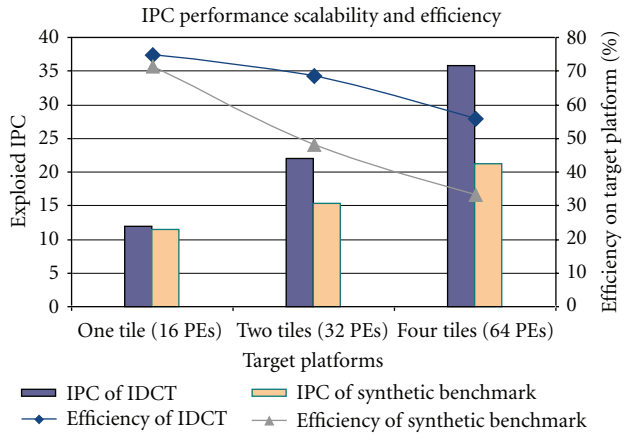



Figure 8: Performance scalability for IDCT and a synthetic benchmark.

bypassing registers that are dedicated to data relay as it does with non-transient copy disabled.

Table 3 compares the performance results for other applications with the feature of non-transient copy on and off. On average, using non-transient copy can improve the IPC by 18%. Two special cases where the IPC does not improve much are idct and sad16_c. This is because these two algorithms are designed in the manner of "in-place-update," in which an instruction replaces the source register with the new result immediately after computation. With this special coding style, registers are used very locally and few operands can be reused. In contrast, the other applications make significant reuse of operands. For example, the computation of two adjacent pixels usually can share a large amount of reference pixels. As a result, distributing the shared reference pixels to multiple PEs as non-transient copies helps for data reusing, and the performance improves significantly.

From Table 3, we can also observe that the total usage of inter-PE links is reduced by 15% on average, which can directly be translated into power saving. Moreover, for most of the applications, both the average and peak bypassing register usage are smaller when non-transient copy is enabled.

*4.4. Performance Scalability Study.* It can be observed from the results in Section 4.2 that the benchmark applications achieved high IPC on both small configuration and large configuration. Actually these video applications can achieve very good performance scalability because of the weak data dependency between each of their kernel iterations. This is illustrated in Figure 9(a), where intra-iteration data dependency is strong while inter-iteration data dependency is weak. As a result, these almost independent iterations can be scheduled across the whole reconfigurable array without incurring much performance penalty. However, it would be interesting to study the performance scalability if an application does not possess this weak inter-iteration property. Unfortunately, good kernels are more than often written in loop fashion with weak inter-iteration data dependency. So we have to create a somewhat contrived synthetic benchmark to simulate what would happen if a kernel has strong inter-iteration dependency. This is done by generating instructions with data dependency with each other for the synthetic benchmark. In order to compare the performance scalability difference between a typical IDCT (row+col) with weak inter-iteration dependency and the synthetic benchmark, we produce the synthetic benchmark to contain the same number of instructions of the IDCT and similar level of instruction-level parallelism but with strong inter-iteration dependency as illustrated in Figure 9(b).

The performance results are plotted in Figure 8 for compiling these two applications on three configurations: one tile, two tile, and four tiles. It is clear that when the target platform has more PEs, IDCT scales better than the synthetic benchmark though they both get performance boost with more available PEs. Specifically, in four-tile configuration where 64 PEs are available, >35 IPC can be achieved in the case of IDCT, but only 21 IPC can be achieved for the synthetic benchmark. Another interesting point about the CGRA efficiency can be observed from Figure 8. Both applications experience efficiency drop when the number of available PEs becomes larger. This is because scattering an application across a large CGRA tends to introduce more operation "bubbles" in the final execution trace. Therefore, the efficiency may degrade with the increasing of PE numbers.

For the synthetic benchmark, not only that the performance does not scale as well as IDCT but also that the efficiency drops more than IDCT. We think this is mainly due to the difference between the data dependency patterns in these two applications. Since IDCT has weak inter-iteration dependency the operations only need to talk to other operations in the same iteration on neighboring PEs locally. In contrast, the strong inter-iteration data dependency pattern in the synthetic benchmark requires more global communication from one PE to another in a different tile. Therefore, the larger the scale of the PE array, the longer distance for transferring data between remote PEs. Consider a scenario in Figure 9(c) for the synthetic benchmark. Assume $M$ is the computation grid (analogous to a metropolis area) and $N$ is a subgrid within $M$ (the downtown). Two routing requirements are shown. There can be chances that the path connecting node 1 and node 2 will also make use of channels in the subgrid $N$ (downtown). Accumulating the effects, the $N$ (downtown) could have heavy communication (traffic) jam. To avoid congestion, the long routing path will probably be dispersed to the outskirt area. Since long distance across multiple tiles may need more routing cycles, the overall performance will be dragged down by this *downtown effect*.

This suggests that it may not be always a good practice to scatter an application to as many available PEs as possible. For certain applications, instead of having the whole computational array to be mapped for a single application kernel, it is preferable to partition the tile-based FDR-CGRA into several regions. Each region carries out one application and run multiple parallel applications simultaneously. In the synthetic benchmark case, compiling it onto one tile and save the other tiles for other tasks may yield the highest efficiency.

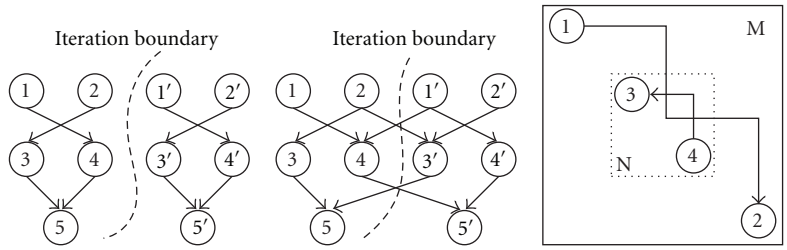



Figure 9: Different benchmark DFGs and downtown effect.

## 5. Conclusions

In this paper, we proposed FDR (fast data relay) to enhance existing coarse-grained reconfigurable computing architectures (CGRAs). FDR utilizes multicycle data transmission and can effectively reduce communication traffic congestion, thus allowing applications to achieve higher performance. To exploit the potential of FDR, a new CAD inspired compilation flow was also proposed to efficiently compile application kernels onto this architecture. It is shown that FDR-CGRA outperformed two other CGRAs, RCP and ADRES, by up to 21% and 29%, respectively.

## References

[1] S. Hauck and A. DeHon, Eds., *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon)*, Morgan Kaufmann, Boston, Mass, USA, 2007.

[2] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings—Computers and Digital Techniques*, vol. 152, no. 2, article 193.

[3] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 12–21, April 1997.

[4] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of the The 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pp. 225–235, June 2000.

[5] R. Hartenstein, "Coarse grain reconfigurable architecture (embedded tutorial)," in *Proceedings of the 16th Asia South Pacific Design Automation Conference (ASP-DAC '01)*, pp. 564–570, 2001.

[6] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD–reconfigur-able pipelined datapath," in *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL '96)*, 1996.

[7] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Matt, and R. R. Taylor, "PipeRench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[8] H. Singh, M. H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[9] R. W. Hartenstein and R. Kress, "Datapath synthesis system for the reconfigurable datapath architecture," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '95)*, pp. 479–484, September 1995.

[10] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '96)*, pp. 157–166, April 1996.

[11] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: a case study," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, pp. 1224–1229, February 2004.

[12] O. Colavin and D. Rizzo, "A scalable wide-issue clustered VLIW with a reconfigurable interconnect," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '03)*, pp. 148–158, November 2003.

[13] M. B. Taylor, W. Lee, J. Miller et al., "Evaluation of the raw microprocessor: an exposed-wire-delay architecture for ILP and streams," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, pp. 2–13, June 2004.

[14] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: an architecture-adaptive CGRA mapping tool," in *Proceedings of the 7th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '09)*, pp. 191–200, February 2009.

[15] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. S. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 166–176, October 2008.

[16] G. Lee, K. Choi, and N. D. Dutt, "Mapping multi-domain applications onto coarse-grained reconfigurable architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 5, pp. 637–650, 2011.

[17] T. Suzuki, H. Yamada, T. Yamagishi et al., "High-throughput, low-power software-defined radio using reconfigurable processors," *IEEE Micro*, vol. 31, no. 6, pp. 19–28, 2011.

[18] Z. Kwok and S. J. E. Wilton, "Register file architecture optimization in a coarse-grained reconfigurable architecture," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 35–44, April 2005.

[19] S. Cadambi and S. C. Goldstein, "Efficient place and route for pipeline reconfigurable architectures," in *Proceedings of*

*the International Conference on Computer Design (ICCD '00)*, pp. 423–429, September 2000.

[20] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA '00)*, pp. 375–386, January 2000.

[21] R. Balasubraamonian, S. Dwarkadas, and D. H. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *Proceedings of the 34th Annual International Symposium on Microarchitecture (ACM/IEEE '01)*, pp. 237–248, December 2001.

[22] B. Mei, F. J. Veredas, and B. Masschelein, "Mapping an H.264/AVC decoder onto the adres reconfigurable architecture," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 622–625, August 2005.

[23] C. Lattner, "Introduction to the LLVM Compiler Infrastructure," in *Itanium Conference and Expo*, April 2006.

[24] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, chapter 3, Morgan Kauffmann, Boston, Mass, USA, 4th edition, 2006.

[25] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[26] R. Nair, "A simple yet effective technique for global wiring," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 2, pp. 165–172, 1987.

[27] Xvid video codec, http://www.xvid.org/.

[28] Opensource H.264 reference code, http://iphome.hhi.de/suehring/tml/.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 201378, 19 pages
doi:10.1155/2012/201378

*Research Article*

# A Protein Sequence Analysis Hardware Accelerator Based on Divergences

**Juan Fernando Eusse,[1] Nahri Moreano,[2] Alba Cristina Magalhaes Alves de Melo,[3] and Ricardo Pezzuol Jacobi[4]**

[1] *Electrical Engineering Department, University of Brasilia, Brasilia, DF 70910-900, Brazil*
[2] *School of Computing, Federal University of Mato Grosso do Sul, Campo Grande, MS 79070-900, Brazil*
[3] *Computer Science Department, University of Brasilia, Brasilia, DF 70910-900, Brazil*
[4] *UnB Gama School, University of Brasilia, Gama, DF 72405-610, Brazil*

Correspondence should be addressed to Ricardo Pezzuol Jacobi, jacobi@unb.br

Received 27 September 2011; Accepted 26 December 2011

Academic Editor: Khaled Benkrid



The Viterbi algorithm is one of the most used dynamic programming algorithms for protein comparison and identification, based on hidden markov Models (HMMs). Most of the works in the literature focus on the implementation of hardware accelerators that act as a prefilter stage in the comparison process. This stage discards poorly aligned sequences with a low similarity score and forwards sequences with good similarity scores to software, where they are reprocessed to generate the sequence alignment. In order to reduce the software reprocessing time, this work proposes a hardware accelerator for the Viterbi algorithm which includes the concept of divergence, in which the region of interest of the dynamic programming matrices is delimited. We obtained gains of up to 182x when compared to unaccelerated software. The performance measurement methodology adopted in this work takes into account not only the acceleration achieved by the hardware but also the reprocessing software stage required to generate the alignment.

## 1. Introduction

Protein sequence comparison and analysis is a repetitive task in the field of molecular biology, as is needed by biologists to predict or determine the function, structure, and evolutional characteristics of newly discovered protein sequences. During the last decade, technological advances had made possible the identification of a vast number of new proteins that have been introduced to the existing protein databases [1, 2]. With the exponential growth of these databases, the execution times of the protein comparison algorithms also grew exponentially [3], and the necessity to accelerate the existing software rose in order to speed up research.

The HMMER 2.3.2 program suite [4] is one of the most used programs for sequence comparison. HMMER takes multiple sequence alignments of similar protein sequences grouped into protein families and builds hidden Markov models (HMMs) [5] of them. This is done to estimate statistically the evolutionary relations that exist between different members of the protein family, and to ease the identification of new family members with a similar structure or function. HMMER then takes unclassified input sequences and compares them against the generated HMMs of protein families (profile HMM) via the Viterbi algorithm (see Section 2), to generate both a similarity score and an alignment for the input (query) sequences.

As the Viterbi routine is the most time consuming part of the HMMER programs, multiple attempts to optimize and accelerate it have been made. MPI-HMMER [6] explores parallel execution in a cluster as well as software optimizations via the Intel-SSE2 instruction set. Other approaches like SledgeHMMER [7] and "HMMER on the Sun Grid" [8] provide web-based search interfaces to either an optimized version of HMMER running on a web server or the Sun Grid, respectively. Other approaches such as ClawHMMER [9] and GPU-HMMER [10] implement GPU parallelization

of the Viterbi algorithm, while achieving a better cost/benefit relation than the cluster approach.

Studies have also shown that most of the processing time of the HMMER software is spent into processing poor scoring (nonsignificant) sequences [11], and most authors have found useful to apply a first-phase filter in order to discard poor scoring sequences prior to full processing. Some works apply heuristics [12], but the mainstream focuses on the use of FPGA-based accelerators [3, 11, 13–16] as a first-phase filter. The filter retrieves the sequence's similarity score and, if it is acceptable, instructs the software to reprocess the sequence in order to generate the corresponding alignment.

Our work proposes further acceleration of the algorithm by using the concept of divergence in which full reprocessing of the sequence after the FPGA accelerator is not needed, since the alignment only appears in specific parts of both the profile HMM model and the sequence. The proposed accelerator outputs the similarity score and the limits of the area of the dynamic programming (DP) matrices that contains the optimal alignment. The software then calculates only that small area of the DP matrices for the Viterbi algorithm and returns the same alignment as the unaccelerated software.

The main contributions of this work are threefold. First, we propose the Plan7-Viterbi divergence algorithm, which calculates the area in the Plan7-Viterbi dynamic programming matrices that contains the sequence-profile alignment. Second, we propose an architecture that implements this algorithm in hardware. Our architecture not only is able to generate the score for a query sequence when compared to a given profile HMM but also generates the divergence algorithm coefficients in hardware, which helps to speed up the subsequent alignment generation process by software. To the best of our knowledge, there is no software adaptation of the divergence algorithm to the Viterbi-Plan7 algorithm nor a hardware implementation of that adaptation. Finally, we propose a new measurement strategy that takes into account not only the architecture's throughput but also reprocessing times. This strategy helps us to give a more realistic measure of the achieved gains when including a hardware accelerator into the HMMER programs.

This work is organized as follows. In Section 2 we clarify some of the concepts of protein sequences, protein families, and profile HMMs. In Section 3 we present the related work in FPGA-based HMMER accelerators. Section 4 introduces the concept of divergences and their use in the acceleration of the Viterbi algorithm. Section 5 shows the proposed hardware architecture. Section 6 presents the metrics used to analyze the performance of the system. In Section 7 we show implementation and performance results, and we compare them with the existing works. Finally, in Section 8 we summarize the results and suggest future works.

## 2. Protein Sequence Comparison

*2.1. Protein Sequences, Protein Families, and Profile HMMs.* Proteins are basic elements that are present in every living organism. They may have several important functions such as catalyzing chemical reactions and signaling if a gene must be expressed, among others. Essentially, a protein is a chain of amino acids. In the nature, there are 20 different amino acids, represented by the alphabet $\Sigma = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$ [17].

A protein family is defined to be a set of proteins that have similar function, have similar 2D/3D structure, or have a common evolutionary history [17]. Therefore, a newly sequenced protein is often compared to several known protein families, in search of similarities. This comparison usually aligns the protein sequence to the representation of a protein family. This representation can be a profile, a consensus sequence, or a signature [18]. In this paper, we will only deal with profile representations, which are based on multiple sequence alignments.

Given a multiple-sequence alignment, a profile specifies, for each column, the frequency that each amino acid appears in the column. If a sequence-profile comparison results in high similarity, the protein sequence is usually identified to be a member of the family. This identification is a very important step towards determining the function and/or structure of a protein sequence.

One of the most accepted probabilistic models to do sequence-profile comparisons is based on hidden Markov models (HMMs). It is called profile HMM because it groups the evolutionary statistics for all the family members, therefore "profiling" it. A profile HMM models the common similarities among all the sequences in a protein family as discrete states; each one corresponding to an evolutionary possibility such as amino acid insertions, deletions, or matches between them. The traditional profile HMM architecture proposed by Durbin et al. [5] consisted of insert ($I$), delete ($D$), and match ($M$) states.

The HMMER suite [4], is a widely used software implementation of profile HMMs for biological sequence analysis, composed of several programs. In particular, the program *hmmsearch* searches a sequence database for matches to an HMM, while the program *hmmpfam* searches an HMM database for matches to a query sequence.

HMMER uses a modified HMM architecture that in addition to the traditional *M, I*, and *D* states includes flanking states that enable the algorithm to produce global or local alignments, with respect to the model or to the sequence, and also multiple-hit alignments [4, 5]. The Plan7 architecture used by HMMER is shown in Figure 1. Usually, there is one match state for each consensus column in the multiple alignment. Each $M$ state aligns to (emits) a single residue, with a probability score that is determined by the frequency in which the residues have been observed in the corresponding column of the multiple alignment. Therefore, each $M$ state has 20 probabilities for scoring the 20 amino acids.

The insertion ($I$) and deletion ($D$) states model gapped alignments, that is, alignments including residue insertions and deletions. Each $I$ state also has 20 probabilities for scoring the 20 amino acids. The group of *M, I*, and *D* states corresponding to the same position in the multiple alignment is called a node of the HMM. Beside the emission

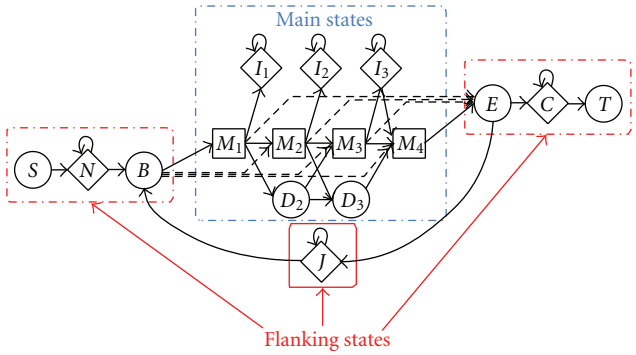



Figure 1: Plan7 architecture used by HMMER [3].

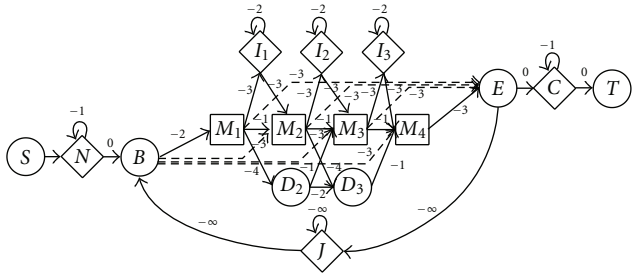



Figure 2: A profile HMM with 4 nodes and the transition scores.

probabilities, there are transition probabilities associated to each transition from one state to another.

*2.2. Viterbi Algorithm.* Given a HMM modeling a protein family and a query sequence, HMMER computes the probability that the sequence belongs to the family, as a similarity score, and generates the resulting alignment if the score is sufficiently good. To do so, it implements a well-known DP algorithm called the Viterbi algorithm [19]. This algorithm calculates a set of score matrices (corresponding to states *M*, *I*, and *D*) and vectors (corresponding to states *N*, *B*, *E*, *C*, and *J*) by means of a set of recurrence equations. As a result, it finds the best (most probable) alignment and its score for the query sequence with the given model.

Equations (1) show the Viterbi algorithm recurrence relations for aligning a sequence of length $n$ to a profile HMM with $k$ nodes. In these equations, $M(i, j)$ is the score of the best path aligning the subsequence $s_1 \ldots s_i$ to the submodel up to state $M_j$ and $I(i, j)$ and $D(i, j)$ are defined similarly. The emission probability of the amino acid $s_i$ at state$_1$ is denoted by em(state$_1$,$s_i$), while tr(state$_1$,state$_2$) represents the transition cost from state$_1$ to state$_2$. The similarity score of the best alignment is given by $C(n) + \mathrm{tr}(C, T)$.

Plan7-Viterbi algorithm recurrence equations, for a profile HMM with $k$ nodes and sequence $s$ of length $n$ are as follows;

$$M(i,0) = I(i,0) = D(i,0) = -\infty \quad \forall 1 \le i \le n,$$

$$M(0,j) = I(0,j) = D(0,j) = -\infty \quad \forall 0 \le j \le k,$$

$$M(i,j) = \mathrm{em}\left(M_j, s_i\right) + \max\begin{cases} M(i-1,j-1) + \mathrm{tr}\left(M_{j-1}, M_j\right) \\ I(i-1,j-1) + \mathrm{tr}\left(I_{j-1}, M_j\right) \\ D(i-1,j-1) + \mathrm{tr}\left(D_{j-1}, M_j\right) \\ B(i-1) + \mathrm{tr}\left(B, M_j\right) \end{cases} \quad \forall 1 \le i \le n,$$

$$I(i,j) = \mathrm{em}\left(I_j, s_i\right) + \max\begin{cases} M(i-1,j) + \mathrm{tr}\left(M_j, I_j\right) \\ I(i-1,j) + \mathrm{tr}\left(I_j, I_j\right) \end{cases} \quad \forall 1 \le j \le k,$$

$$D(i,j) = \max\begin{cases} M(i,j-1) + \mathrm{tr}\left(M_{j-1}, D_j\right) \\ D(i,j-1) + \mathrm{tr}\left(D_{j-1}, D_j\right), \end{cases}$$

$$N(0) = 0,$$

$$N(i) = N(i-1) + \mathrm{tr}(N,N), \quad \forall 1 \le i \le n,$$

$$B(0) = \mathrm{tr}(N,B),$$

$$B(i) = \max\begin{cases} N(i) + \mathrm{tr}(N,B) \\ J(i) + \mathrm{tr}(J,B) \end{cases} \quad \forall 1 \le i \le n,$$

$$E(i) = \max_{1 \le j \le k} \left(M(i,j) + \mathrm{tr}\left(M_j, E\right)\right) \quad \forall 1 \le j \le k,$$

$$J(0) = -\infty,$$

$$J(i) = \max\begin{cases} J(i-1) + \mathrm{tr}(J,J) \\ E(i) + \mathrm{tr}(E,J) \end{cases} \quad \forall 1 \le i \le n,$$

$$C(0) = -\infty,$$

$$C(i) = \max\begin{cases} C(i-1) + \mathrm{tr}(C,C) \\ E(i) + \mathrm{tr}(E,C) \end{cases} \quad \forall 1 \le i \le n.$$

$$\text{similarity\_score} = C(n) + \mathrm{tr}(C,T). \tag{1}$$

Figure 2 illustrates a profile HMM with 4 nodes representing a multiple-sequence alignment. The transition scores are shown in the figure, labeling the state transitions. The emission scores for the $M$ and $I$ states are shown in Table 1.

Table 2 shows the score matrices and vectors computed by the Viterbi algorithm, while aligning the query sequence ACYDE to the profile HMM given in Figure 2. The best alignment has the similarity score of 25 and corresponds to

Table 1: Emission scores of amino acids for match and insert states of profile HMM of Figure 2.

| State | $A$ | $C$ | $D$ | $E$ | $F, I, L, M, V, W$ | $G, K, P, S$ | $H, Q, R, T$ | $Y$ |
|---|---|---|---|---|---|---|---|---|
| $M_1$ | 7 | −1 | −1 | 1 | −1 | 2 | 1 | −1 |
| $M_2$ | −1 | 9 | −1 | 1 | −1 | 2 | 1 | −1 |
| $M_3$ | −1 | −1 | 8 | 2 | −1 | 2 | 1 | −1 |
| $M_4$ | −1 | −1 | 3 | 9 | −1 | 2 | 1 | −1 |
| $I_1$ | −1 | −1 | 0 | 1 | −1 | 0 | 1 | 2 |
| $I_2$ | −1 | −1 | 0 | 1 | −1 | 0 | 1 | 2 |
| $I_3$ | −1 | −1 | 0 | 1 | −1 | 0 | 1 | 2 |

Table 2: Score matrices and vectors of the Viterbi algorithm for the comparison of the sequence ACYDE against the profile HMM of Figure 2.

| | $N$ | $B$ | $M$ | | | | | $I$ | | | | | $D$ | | | | | $E$ | $J$ | $C$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 | | | |
| — | 0 | 0 | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ |
| $A$ | −1 | −1 | −∞ | −5 | −4 | −4 | −4 | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | −∞ | 1 | −1 | −3 | 2 | −∞ | 2 |
| $C$ | −2 | −2 | −∞ | −4 | 13 | −1 | −3 | −∞ | 1 | −8 | −8 | −8 | −∞ | −∞ | −8 | 9 | 7 | 10 | −∞ | 10 |
| $Y$ | −3 | −3 | −∞ | −5 | −3 | 11 | 7 | −∞ | 1 | 12 | 2 | −4 | −∞ | −∞ | −9 | −7 | 7 | 8 | −∞ | 9 |
| $D$ | −4 | −4 | −∞ | −6 | −3 | 17 | 13 | −∞ | −1 | 10 | 8 | 6 | −∞ | −∞ | −10 | −7 | 13 | 14 | −∞ | 14 |
| $E$ | −5 | −5 | −∞ | −5 | −3 | 9 | 25 | −∞ | −2 | 9 | 15 | 24 | −∞ | −∞ | −9 | −7 | 5 | 25 | −∞ | 25 |

the path $(S,\text{-}) \rightarrow (N,\text{-}) \rightarrow (B,\text{-}) \rightarrow (M1,A) \rightarrow (M2,C) \rightarrow (I2,Y) \rightarrow (M3,D) \rightarrow (M4,E) \rightarrow (E,\text{-}) \rightarrow (C,\text{-}) \rightarrow (T,\text{-})$.

## 3. Related Work

The function that implements the Viterbi algorithm in the HMMER suite is the most time consuming of the *hmmsearch* and *hmmpfam* programs of the suite. Therefore, most works [3, 11, 13–16, 20] focus on accelerating its execution by proposing a pre-filter phase which only calculates the similarity score for the algorithm. Then, if the similarity score is good, the entire query sequence is reprocessed to produce the alignment.

In general, FPGA-based accelerators for the Viterbi algorithm are composed of processing elements (PEs), connected together in a systolic array to exploit parallelism by eliminating the $J$ state of the Plan7 Viterbi algorithm (Section 2.2). Usually, each node in the profile HMM is implemented by one PE. However, since the typical profile HMMs contain more than 600 nodes, even the recent FPGAs cannot accommodate this huge number of processing elements. For this reason, the entire sequence processing is divided into several passes [3, 11, 13, 14].

First-in first-out memories are included inside the FPGA implementation to store the necessary intermediary data between passes. Transition and emission probabilities for all the passes of the HMM are preloaded into block memories inside the FPGA to hide model turn around (transition probabilities reloading) when switching between passes. These memory requirements impose restrictions on the maximum PE number that can fit into the device, the maximum HMM size, and the maximum sequence size.

Benkrid et al. [13] propose an array of 90 PEs, capable of comparing a 1024 element sequence with a profile HMM containing 1440 nodes. They eliminate the $J$ state dependencies in order to exploit the dynamic programming parallelism and calculate one cell element per clock cycle in each PE, reporting a maximum performance of 9 GCUPS (giga cell updates per second). Their systolic array was synthesized into a Virtex 2 Pro FPGA with a 100 MHz clock frequency.

Maddimsetty et al. [11] enhance accuracy by reducing the precision error induced by the elimination of the $J$ state and proposes a two-pass architecture to detect and correct false negatives. Based on technology assumptions, they report an estimated maximum size of 50 PEs at an estimated clock frequency of 200 MHz and supposing a performance of 5 to 20 GCUPS.

Jacob et al. [3] divide the PE into 4 pipeline stages, in order to increase the maximum clock frequency up to 180 MHz and the throughput up to 10 GCUPS. Their work also eliminates the $J$ state. The proposed architecture was implemented in a Xilinx Virtex 2 6000 and supports up to 68 PEs, a HMM with maximum length of 544 nodes, and a maximum sequence size of 1024 amino acids.

In Derrien and Quinton [16], a methodology to implement a pipeline inside the PE is outlined, based on the mathematical representation of the algorithm. Then a design space exploration is made for a Xilinx Spartan 3 4000, with maximum PE clock frequency of 66 MHz and a maximum performance of about 1.3 GCUPS.

Oliver et al. [14] implement the typical PE array without taking into account the $J$ state when calculating the score. They obtain an array of 72 PEs working at a clock rate of 74 MHz, and an estimated performance of 3.95 GCUPS.

In [20] a special functional unit is introduced to detect when the $J$ state feedback loop is taken. Then a control unit updates the value for state $B$ and instructs the PEs to

recalculate the inaccurate values. The implementation was made in a Xilinx Virtex 5 110-T FPGA with a maximum of 25 PEs and operating at 130 MHz. The reported performance is 3.2 GCUPS. No maximum HMM length or pass number is reported in the paper.

Takagi and Maruyama [21] developed a similar solution for processing the feedback loop. The alignment is calculated speculatively in parallel, and, when the feedback loop is taken, the alignment is recalculated from the beginning using the feedback score. With a Xilinx XC4VLX160 they could implement 100 PEs for profiles not exceeding 2048 nodes, reaching speedups up to 360 when compared to an Intel Core 2 Duo, 3.16 Ghz, and 4 GB RAM, when no recalculation occurs, and with a corresponding speed-up reduction otherwise.

Walters et al. [15] implement a complete Plan7-Viterbi algorithm in hardware, by exploiting the inherent parallelism in processing different sequences against the same HMM at the same time. Their PE is slightly more complex than those of other works as it includes the *J* state in the score calculation process. They include hardware acceleration into a cluster of computers, in order to further enhance the speedup. The implementation was made in a Xilinx Spartan 3 1500 board with a maximum of 10 PEs per node and a maximum profile HMM length of 256. The maximum clock speed for each PE is 70 MHz, and the complete system yields a performance of 700 MCUPS per cluster node, in a cluster comprised of 10 worker nodes. As any of the other analyzed works, its only output is the sequence score, and for the trace back, a complete reprocessing of the sequence has to be done in software.

Like all the designs discussed in this section, our design does not calculate the alignment in hardware, providing the score as output. Nevertheless, unlike the previous FPGA proposals, our design also provides information that can be used by the software to significantly reduce the number of cells contained in the DP matrices that need to be recalculated. Therefore, beside the score, our design outputs also the divergence algorithm information that will be used by the software to determine a region in the DP matrices where the actual alignment occurs. In this way, the software reprocessing time can be reduced, and better overall speedups can be attained.

Our work also proposes the use of a more accurate performance measurement that includes not only the time spent calculating the sequence score and divergence but also the time spent while reprocessing the sequences of interest, which gives a clearer idea of the real gain achieved when integrating the accelerator to HMMER.

## 4. Plan7-Viterbi Divergence Algorithm

The divergence concept was first introduced by Batista et al. [22], and it was included into an exact variation of the Smith-Waterman algorithm for pairwise local alignment of DNA sequences. Their goal was to obtain the alignment of huge biological sequences, handling the quadratic space, and time complexity of the Smith-Waterman algorithm. Therefore, they used parallel processing in a cluster of processors to reduce execution time and exploited the divergence concept to reduce memory requirements. Initially, the whole similarity matrix is calculated in linear space. This phase of the algorithm outputs the highest similarity score and the coordinates in the similarity matrix that define the area that contains the optimal alignment. These coordinates were called superior and inferior divergences. To obtain the alignment itself using limited memory space, they recalculate the similarity matrix, but this time only the cells inside the limited area need to be computed and stored.

A direct adaptation of the original divergence concept to the Plan7-Viterbi algorithm is not possible because the recurrence relations of the Smith-Waterman and Plan7-Viterbi are totally distinct. The Smith-Waterman algorithm with affine gap calculates three DP matrices (*E*, *F*, *D*), but the inferior and superior divergence could be inferred from only one matrix (*D*) [22]. In the Plan7-Viterbi algorithm (Section 2.2), the inferior and superior divergence information depend on matrices *M*, *I*, *D* and vectors *C*, *E*. For this reason, we had to generate entirely new recurrence relations for divergence calculation. This resulted in a new algorithm, which we called the Plan7-Viterbi divergence algorithm. The recurrence equations for the *M* State of the proposed algorithm are shown in (3) and (4).

Also, the Smith-Waterman divergence algorithm provides a band in the DP matrix, where the alignment occurs, which is limited by the superior and inferior divergences [22]. We observed that the alignment region could be further limited if the initial and final rows are provided, in addition to the superior and inferior divergence information. Therefore, we also extended the divergence concept to provide a polygon that encapsulates the alignment, instead of two parallel lines, as it was defined in the Smith-Waterman divergence algorithm [22]. In the following paragraphs, we describe the Plan7-Viterbi divergence algorithm.

Given the DP matrices of the Viterbi algorithm, the limits of the best alignment are expressed by its initial and final rows and superior and inferior divergences (IR, FR, SD, and ID, resp.). The initial and final rows indicate the row of the matrices where the alignment starts and ends (initial and final element of the sequence involved in the alignment). The superior and inferior divergences represent how far the alignment departs from the main diagonal, in up and down directions, respectively. The main diagonal has divergence 0, the diagonal immediately above it has divergence $-1$, the next one $-2$, and so on. Analogously, the diagonals below the main diagonal have divergences $+1$, $+2$, and so on. These divergences are calculated as the difference $i - j$ between the row ($i$) and column ($j$) coordinates of the matrix cell. Figure 3 shows the main ideas behind the Plan7-Viterbi divergence algorithm.

Given a profile HMM with $k$ nodes and a query sequence of length $n$, the figure shows the DP matrices *M*, *I*, *D* (represented as only one matrix, for clarity) of the Viterbi algorithm. The best alignment of the sequence to the HMM is a path (shown in a thick line) along the cells of the matrices. The initial and final rows of the alignment are 3 and 7, respectively, while the alignment superior and inferior divergences are $-3$ and 0, respectively. These limits

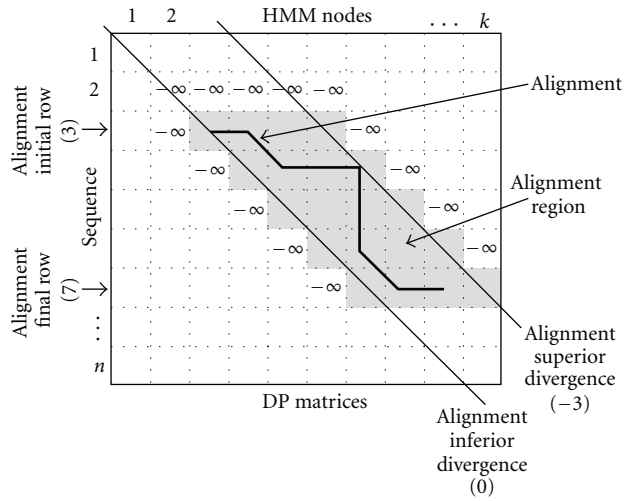



Figure 3: Divergence concept: alignment limits, initialized cells (with $-\infty$), and alignment region, for a HMM with $k$ nodes and a sequence of length $n$.

determine what we define as the alignment region (AR), shown in shadow in the figure.

The AR contains the cells of the score matrices $M$, $I$, and $D$ that must be computed in order to obtain the best alignment. The other Viterbi algorithm DP vectors are also limited by IR and FR, as well. The alignment limits are calculated precisely, leaving no space to error, in a sense that computing only the cells inside the AR will produce the same best alignment as the unbounded (not limited to the AR) computation of the whole matrices.

The Plan7-Viterbi Divergence Algorithm (Plan7-Viterbi-DA) works in two main phases. The first phase is inserted into a simplified version of the Viterbi algorithm which eliminates data dependencies induced by the $J$ state. In this phase, we compute the similarity score of the best alignment of the sequence against the profile HMM, but we do not obtain the alignment itself. We also calculate the limits of the alignment, while computing the similarity score. These limits are computed as new DP matrices and vectors, by means of a new set of recurrence equations. The alignment limits IR, SD, and ID are computed for the *M, I, D, E,* and *C* states. The FR limit is computed only for the *C* state.

The Viterbi algorithm in (1) has the recurrence equation (2) for the $M$ state score computation:

$$M(i,j) = \mathrm{em}\left(M_j, s_i\right) + \max\begin{cases} M(i-1,j-1) + \mathrm{tr}\left(M_{j-1}, M_j\right) \\ I(i-1,j-1) + \mathrm{tr}\left(I_{j-1}, M_j\right) \\ D(i-1,j-1) + \mathrm{tr}\left(D_{j-1}, M_j\right) \\ B(i-1) + \mathrm{tr}\left(B_{i-1}, M_j\right). \end{cases} \quad (2)$$

Let $\mathrm{Sel}_M$ assume the values 0, 1, 2 or 3, depending on the result of the maximum operator in (2). If the argument selected by the maximum operator is the first, second, third, or fourth one, then $\mathrm{Sel}_M$ will assume the value 0, 1, 2, or 3, respectively. Then, the alignment limits IR, SD, and ID, concerning the score matrix $M$, are defined by the recurrence equations in (6).

Recurrence equations for the alignment limits IR, SD, and ID, concerning the score matrix $M$, for $1 \le i \le n$ and $1 \le j \le k$:

$$\mathrm{IR}_M(i,j) = \begin{cases} \mathrm{IR}_M(i-1,j-1), & \text{if } \mathrm{Sel}_M = 0 \\ \mathrm{IR}_I(i-1,j-1), & \text{if } \mathrm{Sel}_M = 1 \\ \mathrm{IR}_D(i-1,j-1), & \text{if } \mathrm{Sel}_M = 2 \\ i, & \text{if } \mathrm{Sel}_M = 3, \end{cases}$$

$$\mathrm{SD}_M(i,j) = \begin{cases} \mathrm{SD}_M(i-1,j-1), & \text{if } \mathrm{Sel}_M = 0 \\ \min(i-j, \mathrm{SD}_I(i-1,j-1)), & \text{if } \mathrm{Sel}_M = 1 \\ \min(i-j, \mathrm{SD}_D(i-1,j-1)), & \text{if } \mathrm{Sel}_M = 2 \\ i-j, & \text{if } \mathrm{Sel}_M = 3, \end{cases}$$

$$\mathrm{ID}_M(i,j) = \begin{cases} \mathrm{ID}_M(i-1,j-1), & \text{if } \mathrm{Sel}_M = 0 \\ \max(i-j, \mathrm{ID}_I(i-1,j-1)), & \text{if } \mathrm{Sel}_M = 1 \\ \max(i-j, \mathrm{ID}_D(i-1,j-1)), & \text{if } \mathrm{Sel}_M = 2 \\ i-j, & \text{if } \mathrm{Sel}_M = 3. \end{cases} \quad (3)$$

The alignment limits IR, SD, and ID, related to the score matrices $I$ and $D$ and vector $E$, are defined analogously, based on the value of $\mathrm{Sel}_I$, $\mathrm{Sel}_D$, and $\mathrm{Sel}_E$, determined by the result of the maximum operator of the Viterbi algorithm recurrence equation for the $I$, $D$, and $E$ states, respectively. Given the recurrence equation for the $C$ state's score computation in the Viterbi algorithm in (1), let $\mathrm{Sel}_C$ assume the values 0 or 1, depending on the result of the maximum operator in this equation. Equation (4) shows the recurrence equations that define the alignment limits IR, FR, SD, and ID, concerning the $C$ score vector.

The first phase of the Plan7-Viterbi-DA was thought to be implemented in hardware because its implementation in software would increase the memory requirements and processing time as it introduces new DP matrices. Besides, the Divergence values computation does not create new data dependencies inside the Viterbi algorithm and can be performed in parallel to the similarity score calculation.

The second phase of the Plan7-Viterbi-DA uses the output data coming from the first one (similarity score and divergence values). If the alignment's similarity score is significant enough, then the second phase generates the alignment. To do this the software executes the Viterbi algorithm again for that sequence.

Nevertheless, it is not necessary to compute the whole DP matrices of the Viterbi algorithm, as we use the alignment limits produced by the first phase in order to calculate only the cells inside the AR of the DP matrices, thus saving memory space and execution time. Figure 4 illustrates

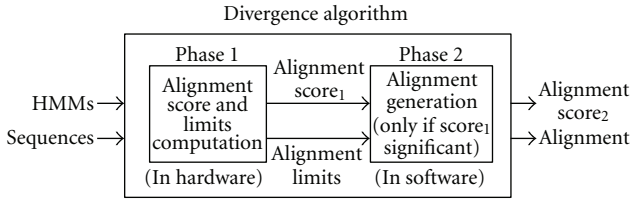



Figure 4: Phases of the Plan7-Viterbi-DA.

the high-level structure of the Plan7-Viterbi-DA and the interaction between its two phases.

Recurrence equations for the alignment limits IR, FR, SD, and ID, concerning the $C$ score vector, for $1 \le i \le n$ :

$$\begin{aligned}
\mathrm{IR}_C(i) &= \begin{cases} \mathrm{IR}_C(i-1), & \text{if } \mathrm{Sel}_C = 0 \\ \mathrm{IR}_E(i), & \text{if } \mathrm{Sel}_C = 1, \end{cases} \\
\mathrm{FR}_C(i) &= \begin{cases} \mathrm{FR}_C(i-1), & \text{if } \mathrm{Sel}_C = 0 \\ i, & \text{if } \mathrm{Sel}_C = 1, \end{cases} \\
\mathrm{SD}_C(i) &= \begin{cases} \mathrm{SD}_C(i-1), & \text{if } \mathrm{Sel}_C = 0 \\ \mathrm{SD}_E(i), & \text{if } \mathrm{Sel}_C = 1, \end{cases} \\
\mathrm{ID}_C(i) &= \begin{cases} \mathrm{ID}_C(i-1), & \text{if } \mathrm{Sel}_C = 0 \\ \mathrm{ID}_E(i), & \text{if } \mathrm{Sel}_C = 1. \end{cases}
\end{aligned} \tag{4}$$

The Plan7-Viterbi-DA's second phase is implemented in software as a modification inside HMMER's Viterbi function used by the *hmmpfam* and *hmmsearch* programs. In this function, we need to initialize with $-\infty$ only the cells immediately above, to the left and to the right of the AR, as shown in Figure 3. The main loops are also modified in order to calculate only the cells inside the AR, using the alignment limits IR, FR, SD, and ID.

In the next section we propose a hardware implementation of the first phase of Plan7-Viterbi-DA.

## 5. HMMER-ViTDiV Architecture

The proposed architecture, called HMMER-ViTDiV, consists of an array of interconnected processing elements (PEs) that implements a simplified version of the Viterbi algorithm, including the necessary modifications to calculate the Plan7-Viterbi-DA presented in Section 4. The architecture is designed to be integrated to the system as a pre-filter stage that returns the similarity score and the alignment limits for a query sequence with a specific profile HMM. If the similarity score for the query sequence is significant enough, then the software uses the alignment limits calculated for the sequence inside the architecture and generates the alignment using the Plan7-Viterbi-DA. Each PE calculates the score for the $j$ column of the DP matrices of the Viterbi algorithm and the alignment limits for the same column. Figure 5 shows the DP matrices antidiagonals and their relationship with each one of the PEs when the number of profile HMM nodes is equal

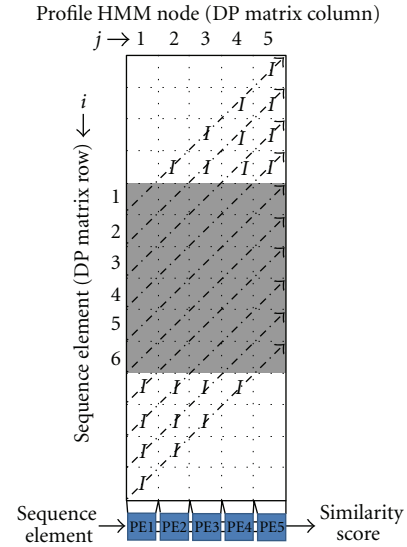



Figure 5: PE to DP matrices correspondence when the HMM number of nodes is less or equal to the number of PEs.

to the number of implemented PEs inside the architecture. In the figure, the arrows show the DP matrices anti-diagonals, cells marked with I correspond to idle PEs, and shaded cells correspond to DP cells that are being calculated by their corresponding PE. The systolic array is filled gradually as the sequence elements are inserted until there are no idle PEs left, and then, when sequence elements are exiting, it empties until there are no more DP cells to calculate.

Since the size of commercial FPGAs is currently limited, today we cannot implement a system with a number of PEs that is equal to one of the largest profile HMM in sequence databases (2295) [2]. We implemented a system that divides the computation into various passes, each one computing a band of size $N$ of the DP matrices, where $N$ is the maximum number of PEs that fits into the target FPGA. In each pass the entire sequence is fed across the array of PEs and the scores are calculated for the current band. Then the output of the last PE of the array is stored inside FIFOs, as it is the input to the next pass and will be consumed by the first PE. Figure 6 presents the concept of band division and multiple passes.

As shown in Figure 6, in each pass the PE acts as a different node of the profile HMM and has to be loaded with the corresponding transition and emission probabilities that are required by the calculations. Also, we note that the system does not have to wait for the entire sequence to be out of the array in one pass to start the next pass, and the PEs can be in different passes at a given time.

Two RAM memories per PE are included inside the architecture to store and provide the transition and emission probabilities for all passes. Two special sequence elements are included in the design to ease the identification of the end of a pass (@) and the end of the sequence processing (*). A controller is implemented inside each PE to identify these two characters, increment or clear the pass number, and signal the transition and emission RAM memories as their address offset depends directly on the pass number.

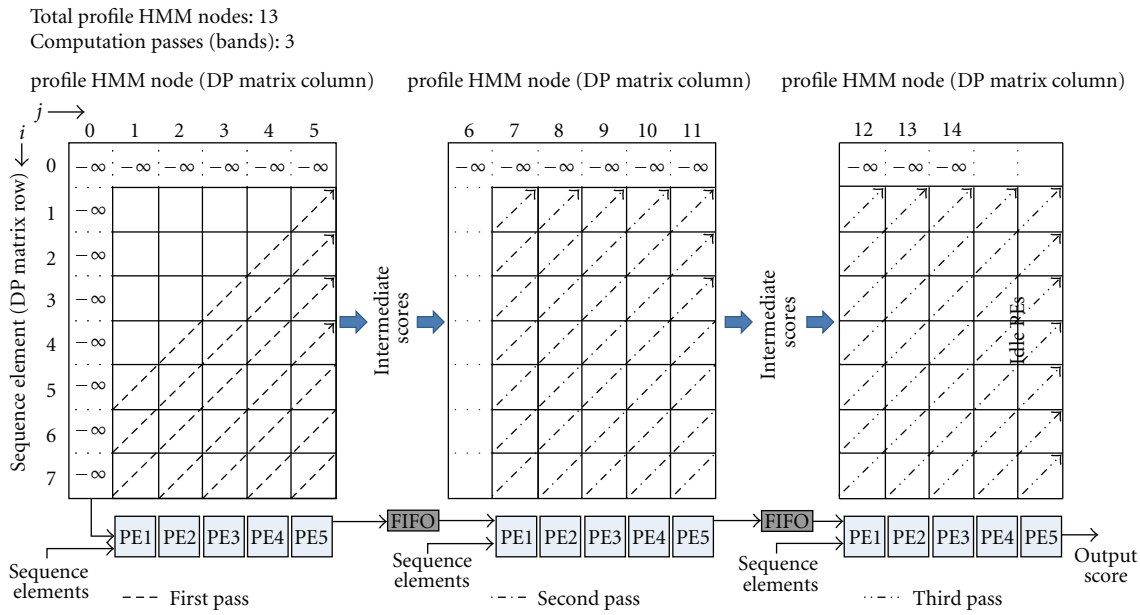



Figure 6: PE to DP matrices correspondence for HMMs with more nodes than the number of PEs (band division and multiple passes).

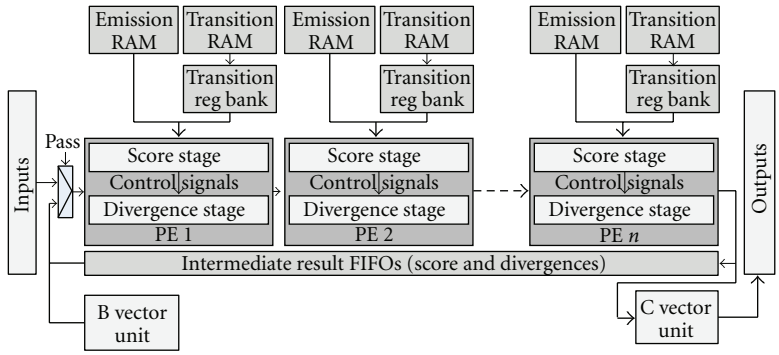



Figure 7: Block diagram of the accelerator architecture.

An input multiplexer had to be included to choose between initialization data for the first pass and intermediate data coming from the FIFOs for the other passes.

A transition register bank had also to be included to store the 9 transition probabilities used concurrently by the PE. This bank is loaded in 5 clock cycles by a small controller inside the transition block RAM memory. Figure 7 shows a general diagram of the architecture.

As illustrated in Figure 7, the PE consists of a score stage which calculates the $M$, $I$, $D$, and $E$ scores and a Plan7-Viterbi divergence stage which calculates the alignment limits for the current sequence. Additional modules are included for the $B$ and $C$ score vector calculations which were placed outside the PE array in order to have an easily modifiable and homogeneous design.

*5.1. Score Stage.* This stage calculates the scores for the $M$, $I$, $D$, and $E$ states of the simplified Viterbi algorithm (without the $J$ state). Each PE represents an individual HMM node, and calculates the scores as each element of the sequence passes through. The PE's inputs are the scores calculated for the current element in a previous HMM node, and the PE's outputs are the scores for the current sequence element in the current node. The score stage of the PE uses (a) 16-bit saturated adders which detect and avoid overflow or underflow errors by saturating the result either to 32767 or to $-32768$ and (b) modified maximum units which not only return the maximum of its inputs but also the index of which of them was chosen. Finally, the score stage consists also of 8 16-bit registers used to store the data required by the DP algorithm to calculate the next cell of the matrix. Figure 8 shows the operator diagram of the score stage. The 4-input maximum unit was implemented in parallel in order to reduce the critical path of the system and thus increase the operating frequency.

*5.2. Plan7-Viterbi Divergence Stage.* This stage calculates the alignment limits for the current query sequence element. The stage inputs are the previous node alignment limits for the current query sequence element, and the outputs are

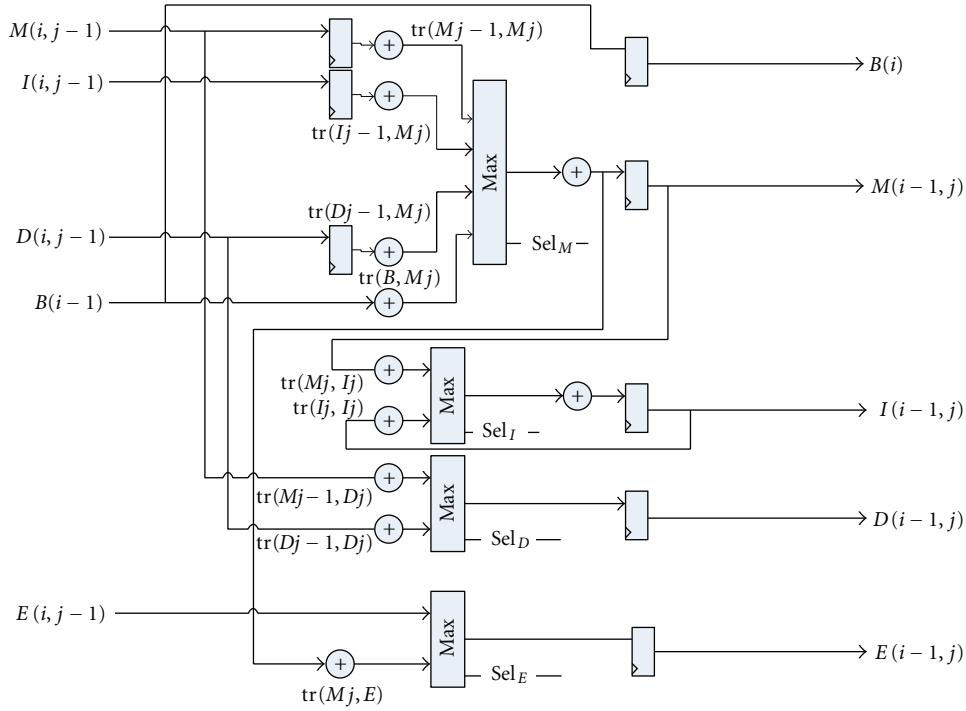



Figure 8: Score stage for the architecture's PE.

the calculated alignment limits for the current element. The outputs depend directly on the score stage of the PE and are controlled by the $\text{Sel}_M$, $\text{Sel}_I$, $\text{Sel}_D$, and $\text{Sel}_E$ signals. The divergence stage also requires the current sequence element index, in order to calculate the alignment limits. Figure 9 shows the Plan7-Viterbi-DA implementation for the *M* and *E* states.

Figures 10 and 11 show the Plan7-Viterbi-DA implementation for the *I* and *D* states, respectively. The Base *J* parameter is the position of the PE in the systolic array, and the #PE parameter is the total number of PEs in the current system implementation. These parameters are used to initialize the divergence stage registers according to the current pass and ensure that the limits are calculated correctly. The divergence stage is composed of (a) 2 input maximum and minimum operators, (b) 4 and 2 input multiplexers, in which the selection lines are connected to the control signals coming from the score stage, and (c) 16-bit registers, which serve as temporal storage for the DP data that is needed to calculate the current divergence DP cell.

*5.3. B and C Score Vector Calculation Units.* The *B* score Vector calculation unit is in charge of feeding the PE array with the *B* score values. This module is placed left of the first PE, and it is connected to the $B(i-1)$ input of it. It has to be initialized for the first iteration with the $\text{tr}(N,B)$ transition probability for the current profile HMM by the control software. For other iterations, it adds the $\text{tr}(N,N)$ probability to the previous values and feeds the output to the first PE. As discussed in Sections 2 and 4, the Plan7-Viterbi-DA does not generate modifications to the *B* calculation unit. Figure 12 shows its hardware implementation.

The *C* calculation unit is in charge of consuming the *E* output provided by the last PE of the array and generating the output similarity score for the current element of the query sequence (the score for the best alignment up to this sequence element). Since the Plan7-Viterbi-DA introduces the calculation of the limits for the best alignment in this state of the Viterbi algorithm, the score stage of the *C* unit also delivers the control signal ($\text{Sel}_C$) for the multiplexers of the divergence stage. Figure 13 shows the *C* state calculation unit, including the score and divergence stages.

## 6. Proposed Performance Measurement

In order to assess the proposed architecture's performance we used two approaches. The first uses the cell updates per second (CUPS) metric, which is utilized by the majority of the previous works [3, 11, 13–16, 20] and measures the quantity of DP matrix cells that the proposed architecture is capable of calculating in one second. We chose this metric in order to compare the performance of our system to the other proposed accelerators. The weakness of the CUPS approach is that it does not consider the reprocessing time and therefore the alignment generation for unaccelerated software, providing an unrealistic measure of the achieved acceleration when integrating the hardware to HMMER.

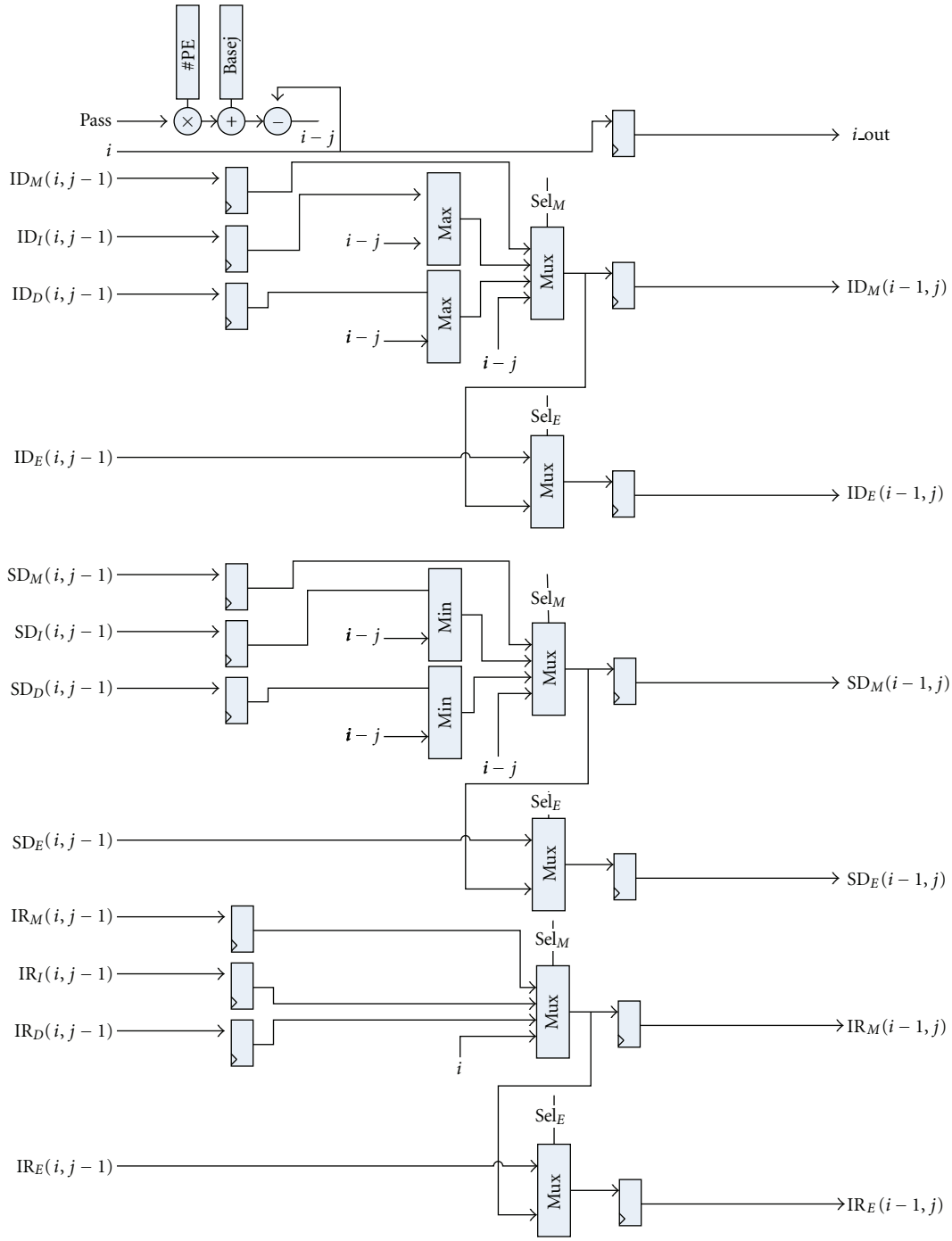



Figure 9: Divergence calculating stage for $M$ and $E$ states.

The second approach measures the execution times of the unaccelerated software when executing a predefined set of sequence comparisons. Then compares it to the execution time of the accelerated system when executing the same set of experiments, to obtain the real gain when integrating a hardware accelerator and the Plan7-Viterbi-DA.

Let $S_t$ be the total number of query sequences in the test set, $P_t$ the total number of profile HMMs in the test set, $t_{s(i,j)}$ the time the unaccelerated *hmmsearch* takes to compare the query sequence $S_i$ to the profile HMM $P_j$, $t_{\text{rep}(i,j)}$ the time the Plan7-Viterbi-DA takes to reprocess the significant query sequence $S_i$ and the profile HMM $P_j$, $t_{\text{con}(i,j)}$ the time spent in communication and control tasks inside the accelerated system, and $t_{h(i,j)}$ the time the hardware accelerator takes to execute the comparison between the query sequence $S_i$ and the profile HMM $P_j$.

Then (5), (6), and (7) show the total time spent by unaccelerated HMMER ($T_{ss}$), the total time spent by the

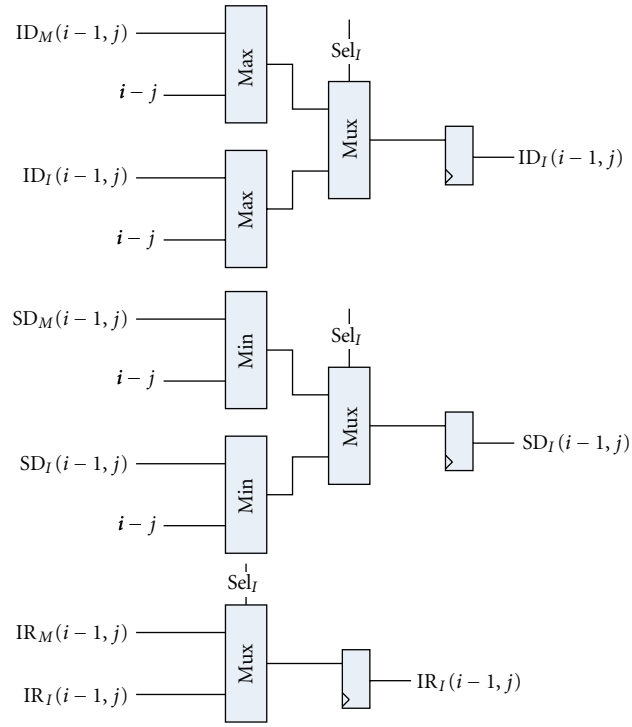



Figure 10: $I$ state divergence calculating stage.

accelerated system ($T_{sa}$), and the achieved performance gains ($G$). The times $t_{s(i,j)}$, $t_{h(i,j)}$, $t_{\mathrm{rep}(i,j)}$, and $t_{\mathrm{con}(i,j)}$ are obtained directly from HMMER, the implemented accelerator, and the software implementing the Plan7-Viterbi-DA and will be shown in the following sections:

$$T_{ss} \longleftarrow \sum_{i=1}^{S_t} \sum_{j=1}^{P_t} t_{s(i,j)}, \tag{5}$$

$$T_{sa} \longleftarrow \sum_{i=1}^{S_t} \sum_{j=1}^{P_t} \left( t_{h(i,j)} + t_{\mathrm{rep}(i,j)} + t_{\mathrm{con}(i,j)} \right), \tag{6}$$

$$G \longleftarrow \frac{T_{ss}}{T_{sa}}. \tag{7}$$

## 7. Experimental Results

The proposed architecture not only enhances software execution by applying a pre-filter to the HMMER software but also provides a means to limit the area of the DP matrices that needs to be reprocessed, by software, in the case of significant sequences. Because of this, the speedup of the solution must be measured by taking into account the performance achieved by the hardware pre-filter as well as the saved software processing time by only recalculating the scores inside the alignment region. Execution time is measured separately for the hardware by measuring its real throughput rate (including loading time and interpass delays) and for software by computing the savings when calculating the scores and the alignment of the divergence-limited region of the DP matrices (Figure 3).

Experimental tests were conducted over all the 10340 profile HMMs for the PFam-A protein database [2]. Searches were made using 4 sets of 2000 randomly sampled protein sequences from the UniProtKB/SwissProt protein database [1] and only significantly scoring sequences were considered to be reprocessed in software. To find out which sequences from the sequence set were significant, we utilized a user-defined threshold and relaxed it to include the greatest possible number of sequences [11]. The experiments were done several times to guarantee the repeatability of them and the stability of the obtained data.

*7.1. Implementation and Synthesis Results.* The complete system was implemented in VHDL and mapped to an Altera Stratix II EP2S180F1508C3 device. Several configurations were explored to maximize the number of HMM nodes, the number of PEs, and the maximum sequence length. In order to do design space exploration, we developed a parameterizable VHDL code, in which we can modify the PE word size, the number of PEs of the array, and the size of the memories.

For the current implementation, we obtained a maximum frequency of 67 MHz after constraining the design time requirements in the Quartus II tool to optimize the synthesis for speed instead of area. Further works will include pipelining the PE to achieve better performance in terms of clock frequency. Table 3 shows the synthesized configurations and their resource utilization.

*7.2. Unaccelerated HMMER Performance.* To measure the *hmmsearch* performance in a typical work environment

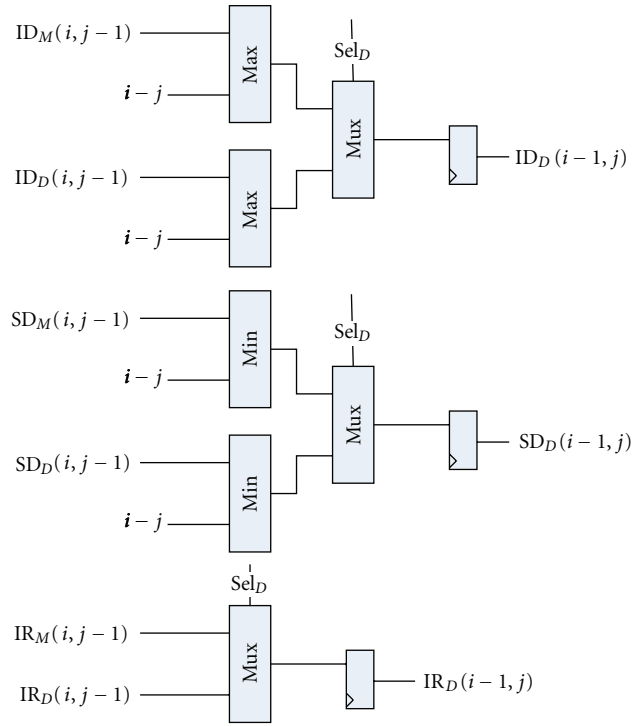



Figure 11: $D$ state divergence calculating stage.

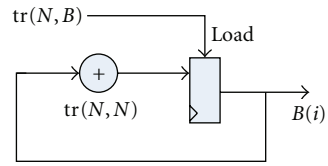



Figure 12: $B$ score vector calculation unit (see Section 2).

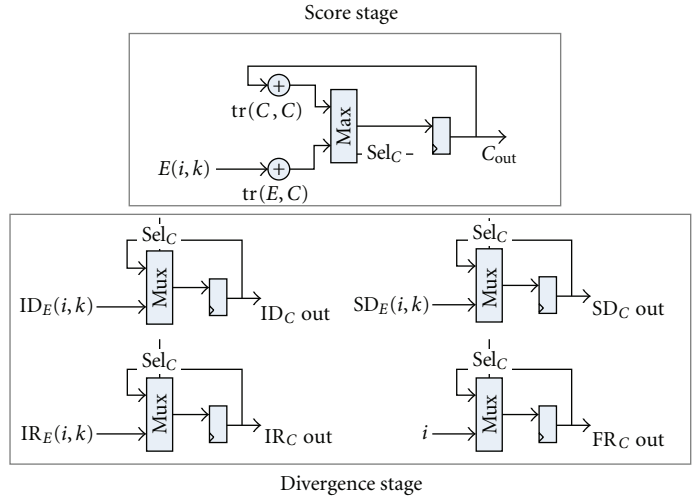



Figure 13: $C$ vector calculation unit.

Table 3: Area and performance synthesis results.

| NO. of PEs | Max. passes | Max. HMM nodes | Max. sequence size | Combinational ALUs | Dedicated registers | Memory bits | % Logic | Max. clock frequency (MHz) |
|---|---|---|---|---|---|---|---|---|
| 25 | 25 | 625 | 8192 | 31738 | 18252 | 2609152 | 25 | 71 |
| 50 | 25 | 1250 | 8192 | 59750 | 35294 | 3121152 | 49 | 71 |
| 75 | 25 | 1875 | 8192 | 93132 | 52520 | 3663152 | 75 | 69 |
| 85 | 27 | 2295 | 8192 | 103940 | 59285 | 5230592 | 84 | 67 |

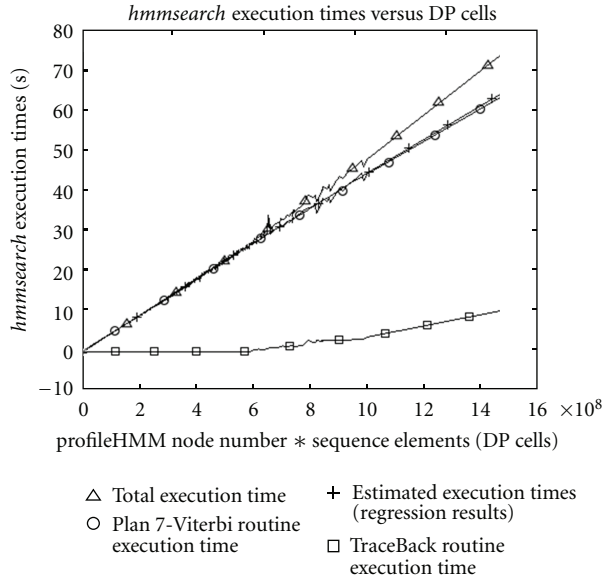



Figure 14: Unaccelerated *hmmsearch* performance for the test set.

we used a platform composed of an Intel Centrino Duo processor running at 1.8 GHz, 4 GB of RAM memory, and a 250 GB hard drive. HMMER was compiled to optimize execution time inside a Kubuntu Linux distribution. We also modified the *hmmsearch* program in order to obtain the execution times only for the Viterbi algorithm, as it was our main target for acceleration.

The characterization of HMMER was done by executing the entire set of tests (4 sets of 2000 randomly sampled sequences compared against 10340 profile HMMs) in the modified *hmmsearch* program. This was done to obtain an exact measure of the execution times of the unaccelerated software and to make its characterization when executing in our test platform. Figure 14 shows the obtained results for the experiments.

The line with triangular markers represents the total execution time of the *hmmsearch* program including the alignment generation times, the line with circular markers represents the execution time only for the Viterbi algorithm, the line with square markers represents the time consumed by the program when generating the alignments, and the line with the plus sign markers corresponds to the expected execution times obtained via the characterization expression shown in (8).

Let $l_i$ be the number of amino acids in sequence $S_i$ and let $m_j$ be the number of nodes in the profile HMM $P_j$. Then the time to make the comparison between the profile HMM and the query sequence ($t_{s(i,j)}$) was found to be accurately represented by (8) which was found by making a least-squares regression on the data plotted in the circle-marked line of Figure 14:

$$t_{s(i,j)} \longleftarrow -1.3684 * 10^{-18}\left(m_j l_i\right)^2 + 4.3208 * 10^{-8} * \left(m_j l_i\right) - 0.1160. \tag{8}$$

Even though we ran our tests with all the profile HMMs in the PFam-A database [2], we chose to show results only for 6 representative profile HMMs that include the smallest and the largest of the database, due to space limitations. Table 4 shows the estimated execution time obtained with (8) and its error percentage when compared to the actually measured execution times. We can calculate the number of average cell updates per second (CUPS) as the total number of elements of the entire data set sequences times the number of nodes of the profiles in the set divided by the complete execution time of the processing. We obtained a performance of 23.157 mega-CUPS for HMMER executing on the test platform.

*7.3. Hardware Performance.* We formulated an equation for performance prediction of the proposed accelerator, taking into account the possible delays, including systolic array data filling and consuming, profile HMM probabilities loading into RAM memories, and probability reloading delays when switching between passes. In order to validate the equation's results, we developed a test bench to execute all the test sets. I/O data transmission delays from/to the PC host were not considered into the formula due to the fact that, in platforms such as the XD2000i [23], data transmission rates are well above the maximum required for the system (130 MBps).

Let $m_i$ be the number of nodes of the current HMM, $S_j$ the size of the current query sequence being processed, $n$ the number of PEs in hardware, $f$ the maximum system frequency, $T_{hw}$ the throughput of the system (measured in CUPS), and $t_{h(i,j)}$ the time the accelerator takes to process one sequence set. Then $T_{hw}$ and $t_{h(i,j)}$ are fully described by (9) and (10), where $25n\lceil m_i/n\rceil$ are the number of cycles spent loading the current HMM into memory, $n$ are the array filling number of cycles, $(S_j + 6)\lceil m_i/n\rceil$ are the cycles spent while processing the current sequence, 3 are the cycles spent loading the special transitions, and $S_j m_i$ are number of cells

Table 4: Modified *hmmsearch* performance results.

| Sequence set elements | Number of HMM nodes | Measured time (total) | Measured time (Viterbi only) | Estimated time (Viterbi only) | Error (%) |
|---|---|---|---|---|---|
| 687406 | 788 | 23.40 | 23.28 | 22.871 | 1.75 |
| | 10 | 0.40 | 0.35 | 0.1810 | 48.2 |
| | 226 | 6.55 | 6.47 | 6.5635 | 1.42 |
| | 337 | 9.85 | 9.8 | 9.8199 | 0.2 |
| | 2295 | 74.49 | 64.09 | 64.6425 | 0.8 |
| | 901 | 26.32 | 26.25 | 26.1199 | 0.4 |
| 697407 | 788 | 24.34 | 23.48 | 23.2158 | 1.12 |
| | 10 | 0.47 | 0.41 | 0.1853 | 54.1 |
| | 226 | 6.68 | 6.66 | 6.6602 | 0.003 |
| | 337 | 9.88 | 9.83 | 9.9634 | 1.35 |
| | 2295 | 78.87 | 62.89 | 65.5334 | 4.2 |
| | 901 | 27.55 | 25.96 | 26.4938 | 2.05 |
| 700218 | 788 | 24.40 | 23.37 | 23.3082 | 0.264 |
| | 10 | 0.42 | 0.38 | 0.1865 | 50.92 |
| | 226 | 6.76 | 6.72 | 6.6873 | 0.486 |
| | 337 | 10.26 | 9.84 | 10.0037 | 1.663 |
| | 2295 | 81.23 | 62.25 | 65.7849 | 5.678 |
| | 901 | 27.41 | 26.33 | 26.5989 | 1.021 |
| 712734 | 788 | 25.42 | 24.03 | 23.7193 | 1.293 |
| | 10 | 0.42 | 0.37 | 0.1919 | 48.13 |
| | 226 | 6.82 | 6.77 | 6.8083 | 0.565 |
| | 337 | 10.09 | 10.01 | 10.1832 | 1.730 |
| | 2295 | 81.07 | 63.81 | 66.8985 | 4.840 |
| | 901 | 27.46 | 26.82 | 27.0665 | 0.919 |

* Execution times are all expressed in seconds.

that the unaccelerated algorithm will have to calculate to process the current sequence with the current HMM:

$$T_{hw} = \frac{\sum_{i=1}^{\#\text{Seqs}} \sum_{j=1}^{\#\text{HMMs}} S_i m_j}{\sum_{j=1}^{\#\text{HMMs}} \left[ \left( \sum_{i=1}^{\#\text{Seqs}} (S_i + 6) \left\lceil m_j/n \right\rceil \right) + 25n \left\lceil m_j/n \right\rceil + n - 2 \right]} * f, \tag{9}$$

$$t_{h(i,j)} = \frac{(S_i + 6) \left\lceil m_j/n \right\rceil + 25n \left\lceil m_j/n \right\rceil + n - 2}{f}. \tag{10}$$

We made the performance evaluation for the 4 proposed systolic PE arrays (25, 50, 75, and 85 PEs) and found out that the two characteristics that greatly influence the performance of the array are the quantity of PEs implemented in the array and the number of nodes of the profile HMM we are comparing the sequences against.

Table 5 shows the obtained performances for all the array variations when executing the comparisons for our 4 sets of sequences against the 6 profile HMM subsets. The best result for each case is shown in bold. From the table we can see that performance increases significantly with the number of implemented PEs. Also we can observe that the system has better performance for profile HMMs whose node number is an exact multiple of the array node number. This is due to the fact that, when a PE does not correspond to a node inside the profile HMM, its transition and emission probabilities are set to minus infinity in order to stop that PE to modify the previously calculated result and only forward that result, thus wasting a clock cycle and affecting performance.

Figures 15 and 16 show the variations in the accelerator performance with the implemented PE number and the profile HMM node number, as seen from the experimental results.

From Figure 16 we can see that, as the performance varies according with profile HMM node number, there is an envelope curve around the performance data which shows the maximum and minimum performances of the array when varying the number of the HMM nodes.

*7.4. Reprocessing Stage Performance (with Plan7-Viterbi-DA).* When aligning different sequences with profile HMMs it is unlikely to find two alignments that are equal. Due to this fact, we cannot predict beforehand what will be the performance of the reprocessing stage as the divergence limits for every alignment are likely to be different. To make

Table 5: Hardware performance results.

| Sequence set elements | Number of HMM nodes | 25 PEs | | 50 PEs | | 75 PEs | | 85 PEs | |
|---|---|---|---|---|---|---|---|---|---|
| | | $T_{hw}$ (GCUPS) | $t_h$ (sec) | $T_{hw}$ (GCUPS) | $t_h$ (sec) | $T_{hw}$ (GCUPS) | $t_h$ (sec) | $T_{hw}$ (GCUPS) | $t_h$ (sec) |
| 687406 | 788 | 1.7468 | 0.3101 | 3.4903 | 0.1552 | 4.9293 | 0.1068 | 5.4203 | 0.0971 |
| | 10 | 0.7093 | 0.0097 | 0.7086 | 0.0097 | 0.6880 | 0.0097 | 0.6877 | 0.0097 |
| | 226 | 1.6031 | 0.0969 | 3.2033 | 0.0485 | 3.8876 | 0.0388 | 5.1815 | 0.0291 |
| | 337 | 1.7075 | 0.1357 | 3.4118 | 0.0679 | 4.6377 | 0.0485 | 5.7949 | 0.0388 |
| | 2295 | **1.7695** | 0.8915 | **3.5358** | 0.4462 | **5.0942** | 0.3010 | **5.8468** | 0.2622 |
| | 901 | 1.7274 | 0.3586 | 3.3607 | 0.1843 | 4.7691 | 0.1262 | 5.6341 | 0.1068 |
| 697407 | 788 | 1.7468 | 0.3146 | 3.4904 | 0.1574 | 4.9295 | 0.1083 | 5.4205 | 0.0985 |
| | 10 | 0.7093 | 0.0098 | 0.7086 | 0.0098 | 0.6880 | 0.0099 | 0.6877 | 0.0099 |
| | 226 | 1.6031 | 0.0983 | 3.2033 | 0.0492 | 3.8878 | 0.0394 | 5.1817 | 0.0296 |
| | 337 | 1.7075 | 0.1376 | 3.4119 | 0.0689 | 4.6379 | 0.0492 | 5.7952 | 0.0394 |
| | 2295 | **1.7695** | 0.9045 | **3.5359** | 0.4527 | **5.0944** | 0.3053 | **5.8471** | 0.2660 |
| | 901 | 1.7274 | 0.3638 | 3.3608 | 0.1870 | 4.7693 | 0.1280 | 5.6344 | 0.1084 |
| 700218 | 788 | 1.7468 | 0.3159 | 3.4905 | 0.1581 | 4.9296 | 0.1088 | 5.4206 | 0.0989 |
| | 10 | 0.7093 | 0.0099 | 0.7086 | 0.0099 | 0.6880 | 0.0099 | 0.6877 | 0.0099 |
| | 226 | 1.6031 | 0.0987 | 3.2034 | 0.0494 | 3.8878 | 0.0396 | 5.1818 | 0.0297 |
| | 337 | 1.7075 | 0.1382 | 3.4120 | 0.0692 | 4.6379 | 0.0494 | 5.7953 | 0.0396 |
| | 2295 | **1.7695** | 0.9081 | **3.5359** | 0.4545 | **5.0945** | 0.3066 | **5.8472** | 0.2671 |
| | 901 | 1.7274 | 0.3652 | 3.3608 | 0.1877 | 4.7693 | 0.1286 | 5.6345 | 0.1088 |
| 712734 | 788 | 1.7468 | 0.3215 | 3.4906 | 0.1609 | 4.9298 | 0.1107 | 5.4209 | 0.1007 |
| | 10 | 0.7093 | 0.0100 | 0.7086 | 0.0101 | 0.6880 | 0.0101 | 0.6878 | 0.0101 |
| | 226 | 1.6031 | 0.1005 | 3.2035 | 0.0503 | 3.8880 | 0.0403 | 5.1821 | 0.0302 |
| | 337 | 1.7075 | 0.1407 | 3.4121 | 0.0704 | 4.6382 | 0.0503 | 5.7956 | 0.0403 |
| | 2295 | **1.7695** | 0.9244 | **3.5360** | 0.4626 | **5.0947** | 0.3120 | **5.8475** | 0.2719 |
| | 901 | 1.7274 | 0.3718 | 3.3609 | 0.1911 | 4.7696 | 0.1308 | 5.6348 | 0.1108 |

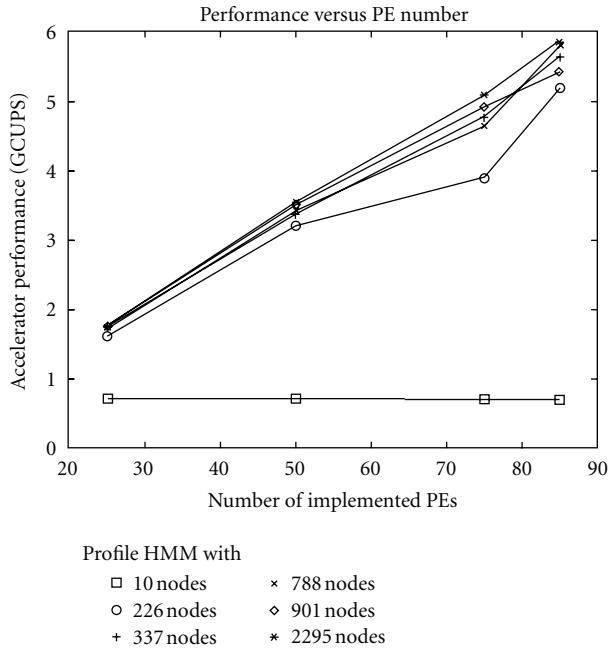



Figure 15: Performance versus number of PEs relation.

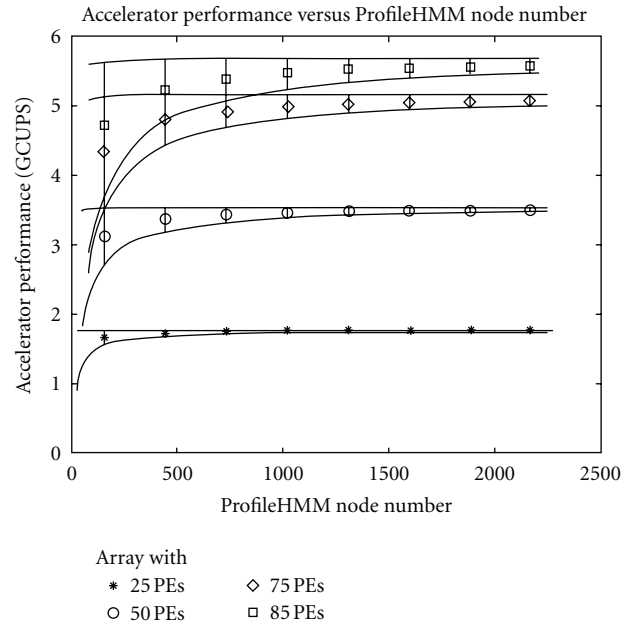



Figure 16: Performance versus number of HMM nodes envelope curves.

Table 6: Second-stage performance estimations.

| Sequence set elements | Number of HMM nodes | Entire sequence set processing time in unaccelerated HMMER (sec) | Significant sequences reprocessing time with prefilter and unaccelerated HMMER (sec) | Divergence accelerated significant sequences reprocessing time (sec) |
|---|---|---|---|---|
| 687406 | 788 | 23.40 | 0.234 | 0.0515 |
| | 10 | 0.40 | 0.004 | 0.0009 |
| | 226 | 6.55 | 0.0655 | 0.0144 |
| | 337 | 9.85 | 0.0985 | 0.0217 |
| | 2295 | 74.49 | 0.7449 | 0.1639 |
| | 901 | 26.32 | 0.2632 | 0.0579 |
| 697407 | 788 | 24.34 | 0.2434 | 0.0535 |
| | 10 | 0.47 | 0.0047 | 0.001 |
| | 226 | 6.68 | 0.0668 | 0.0147 |
| | 337 | 9.88 | 0.0988 | 0.0217 |
| | 2295 | 78.87 | 0.7887 | 0.1735 |
| | 901 | 27.55 | 0.2755 | 0.0606 |
| 700218 | 788 | 24.40 | 0.244 | 0.0537 |
| | 10 | 0.42 | 0.0042 | 0.0009 |
| | 226 | 6.76 | 0.0676 | 0.0149 |
| | 337 | 10.26 | 0.1026 | 0.0226 |
| | 2295 | 81.23 | 0.8123 | 0.1787 |
| | 901 | 27.41 | 0.2741 | 0.0603 |
| 712734 | 788 | 25.42 | 0.2542 | 0.0559 |
| | 10 | 0.42 | 0.0042 | 0.0009 |
| | 226 | 6.82 | 0.0682 | 0.015 |
| | 337 | 10.09 | 0.1009 | 0.0222 |
| | 2295 | 81.07 | 0.8107 | 0.1784 |
| | 901 | 27.46 | 0.2746 | 0.0604 |

an estimate of the performance of the second stage, we made a study in which we executed the comparison of the 20 top profile HMMs from the PFam-A [2] database with our 4 sets of query sequences to obtain both the similarity score and the divergence data for them. Then we built a graph plotting the similarity score threshold and the number of sequences with a similarity score greater than the threshold. From this graph we learned that less than 1% of the sequences were considered significant, even relaxing the threshold to include very bad alignments. With this information, we plotted the percentage of the DP matrices that the second stage of the system will have to reprocess in order to find out the worst case situation and make our estimations based on it. From Figure 17 we can see that, for the experimental data considered, in the worst case the divergence region only corresponds to 22% of the DP matrices.

To obtain the second-stage performance estimations for HMMER ($t_{\text{rep}(i,j)}$ in (6)), we obtained the percentage of significant sequences ($p_s$), multiplied it by the worst case percentage of the DP matrices that the second stage has to reprocess in order to generate the alignment ($p_c$), and then we multiplied it by the time the program *hmmsearch* takes to do the whole query sequence ($S_i$) comparison with a profile HMM ($P_j$). Equation (8) shows the expression used to estimate the performance for the second stage. Table 6 presents the obtained results and also shows the comparison between the times the second stage will spend reprocessing the significant sequences with and without the Plan7-Viterbi-DA. As shown in Table 6, we obtained a performance gain up to 5 times only in the reprocessing stage.

$$t_{(\text{reg}(i,j))} = t * p_s * p_c \tag{11}$$

*7.5. Total System Performance.* In Section 6, we proposed two approaches to evaluate the performance for the system. For the first approach based in CUPS, we obtained a maximum system performance of up to 5.8 GCUPS when implementing a system composed by 85 PEs. This gives us a maximum gain of 254 times over the performance of unaccelerated HMMER software. For the second approach, as we obtained the individual processing times for every stage of the execution, we can determine the overall system performance by applying (6) to the results obtained in Tables 5 and 6. When including the Plan7-Viterbi divergence reprocessing stage, we got a maximum gain of up to 182 times the unaccelerated software, which still means a significant gain when comparing to unaccelerated HMMER. Table 7 presents the total execution time of the system and

Table 7: Total system performance and obtained performance gains.

| Sequence set elements | Number of HMM nodes | Prefilter hardware execution time (sec) | Divergence second-stage execution time (sec) | Total time ($t_{sa(i,j)}$) | Unaccelerated HMMER execution time (sec) | Obtained gain |
|---|---|---|---|---|---|---|
| 687406 | 788 | 0.0971 | 0.0515 | 0.1486 | 23.40 | 157.4697 |
| | 10 | 0.0097 | 0.0009 | 0.0106 | 0.40 | 37.7358 |
| | 226 | 0.0291 | 0.0144 | 0.0435 | 6.55 | 150.5747 |
| | 337 | 0.0388 | 0.0217 | 0.0605 | 9.85 | 162.8099 |
| | 2295 | 0.2622 | 0.1639 | 0.4261 | 74.49 | **174.8181** |
| | 901 | 0.1068 | 0.0579 | 0.1647 | 26.32 | 159.8057 |
| 697407 | 788 | 0.0985 | 0.0535 | 0.152 | 24.34 | 160.1316 |
| | 10 | 0.0099 | 0.001 | 0.0109 | 0.47 | 43.1193 |
| | 226 | 0.0296 | 0.0147 | 0.0443 | 6.68 | 150.7901 |
| | 337 | 0.0394 | 0.0217 | 0.0611 | 9.88 | 161.7021 |
| | 2295 | 0.2660 | 0.1735 | 0.4395 | 78.87 | **179.4539** |
| | 901 | 0.1084 | 0.0606 | 0.169 | 27.55 | 163.0178 |
| 700218 | 788 | 0.0989 | 0.0537 | 0.1526 | 24.40 | 159.8952 |
| | 10 | 0.0099 | 0.0009 | 0.0108 | 0.42 | 38.8889 |
| | 226 | 0.0297 | 0.0149 | 0.0446 | 6.76 | 151.5695 |
| | 337 | 0.0396 | 0.0226 | 0.0622 | 10.26 | 164.9518 |
| | 2295 | 0.2671 | 0.1787 | 0.4458 | 81.23 | **182.2118** |
| | 901 | 0.1088 | 0.0603 | 0.1691 | 27.41 | 162.0934 |
| 712734 | 788 | 0.1007 | 0.0559 | 0.1566 | 25.42 | 162.3244 |
| | 10 | 0.0101 | 0.0009 | 0.011 | 0.42 | 38.1818 |
| | 226 | 0.0302 | 0.015 | 0.0452 | 6.82 | 150.885 |
| | 337 | 0.0403 | 0.0222 | 0.0625 | 10.09 | 161.44 |
| | 2295 | 0.2719 | 0.1784 | 0.4503 | 81.07 | **180.0355** |
| | 901 | 0.1108 | 0.0604 | 0.1712 | 27.46 | 160.3972 |

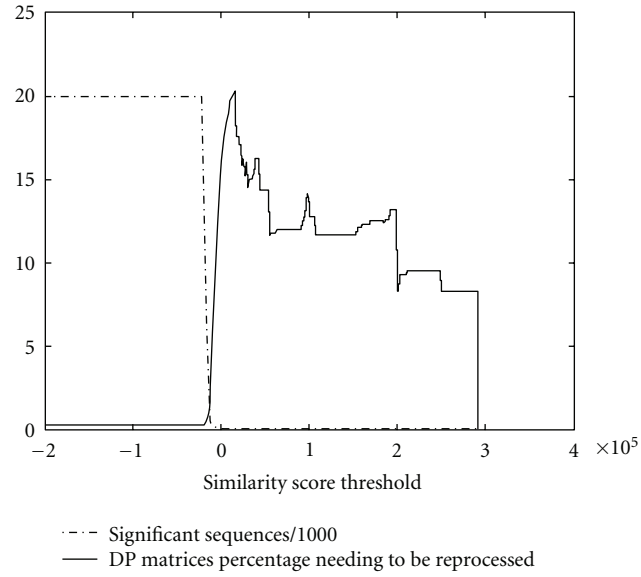



Figure 17: Number of significant sequences and DP percentage that is required to reprocess in software versus similarity score threshold.

Table 8: Related work and comparison with our solution.

| Reference | Number of PEs | Max. number of HMM nodes | Max. sequence size | Complete Plan7-Viterbi algorithm | Clock (MHz) | Performance (GCUPS) | Gain | FPGA |
|---|---|---|---|---|---|---|---|---|
| [3] | 68 | 544 | 1024 | *N* | 180 | 10 | 190 | Xilinx Virtex II 6000 |
| [13] | 90 | 1440 | 1024 | *N* | 100 | 9 | 247 | Xilinx 2VP100 |
| [11] | 50 | — | — | *N* | 200 | 5 to 20 | — | Not Synthesized |
| [14] | 72 | 1440 | 8192 | *N* | 74 | 3.95 | 195 | XC2V6000 |
| [15] | 10 | 256 | — | *Y* | 70 | 7 | 300 | XC3S1500 |
| [16] | 50 | — | — | *N* | 66 | 1.3 | 50 | Xilinx Spartan 3 4000 |
| [20] | 25 | — | — | *Y* | 130 | 3.2 | 56.8 | Xilinx Virtex 5 110-T |
| **Our work** | **85** | **2295** | **8192** | *N* | **67** | **5.8** | **254 (182*)** | **Altera Stratix II EP2S180F1508C3** |

* Including significant sequences reprocessing times.

shows the obtained performance gains. The table presents a summary of the execution time for each individual part of the system and calculates the performance gains with respect to the unaccelerated HMMER by applying (6).

Table 8 shows a brief comparison of this work with the ones found in the literature. We support longest test sequences and bigger profiles when compared with most works. As we include the divergence stage, we have to pay an area penalty, which limit the maximum number of PEs when compared with [13] and can affect the performance of the system. Nevertheless, we obtain additional gains from the divergence, a fact that justify the area overhead. Our architecture performs worse than complete Viterbi implementations such as the one presented in [15], but these cases are uncommon, and even if we relax the threshold for a sequence to be considered significant, the inclusion of the accelerator yields significant speedup.

## 8. Conclusions

In this paper, we introduced the Plan7-Viterbi-DA that enables the implementation of a hardware accelerator for the *hmmsearch* and *hmmpfam* programs of the HMMER suite. We proposed an accelerator architecture which acts as a pre-filtering phase and uses the Plan7-Viterbi-DA to avoid the full reprocessing of the sequence in software. We also introduced a more accurate performance measurement strategy when evaluating HMMER hardware accelerators, which not only includes the time spent on the pre-filtering phase or the hardware throughput but also includes reprocessing times for the significant sequences found in the process.

We implemented our accelerator in VHDL, obtaining performance gains of up to 182 times the performance of the HMMER software. We also made a comparison of the present work with those found in the literature and found that, despite the increased area, we managed to fit a considerable amount of PEs inside the FPGA, which are capable of comparing query sequences with even the largest profile HMM present in the PFam-A database.

For future works we intend to adapt the Plan7-Viterbi-DA to a complete version of the Plan7-Viterbi algorithm (including the *J* state) and make a pipelined version of the PE architecture, in order to further increase the performance gains achieved when integrating the array with the HMMER software.

## Acknowledgments

The authors would like to acknowledge the CNPq, the National Microelectronics Program (PNM), the FINEP, the Brazilian Millennium Institute (NAMITEC), the CAPES, and the Fundect-MS for funding this work.

## References

[1] The Universal Protein Resource—UniProt, June 2009, http://www.uniprot.org/.

[2] Sanger's Institute PFAM Protein Sequence Database, May 2009, http://pfam.sanger.ac.uk/.

[3] A. C. Jacob, J. M. Lancaster, J. D. Buhler, and R. D. Chamberlain, "Preliminary results in accelerating profile HMM search on FPGAs," in *Proceedings of the 21st International Parallel and Distributed Processing Symposium, (IPDPS '07)*, Long Beach, Calif, USA, March 2007.

[4] "HMMER: biosequence analysis using profile hidden Markov models," 2006, http://hmmer.janelia.org/.

[5] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis Probabilistic Models of Proteins and Nucleic Acids*, Cambridge University Press, New York, NY, USA, 2008.

[6] R. Darole, J. P. Walters, and V. Chaudhary, "Improving MPI-HMMER's scalability with parallel I/O," Tech. Rep. 2008-11, Department of Computer Science and Engineering, University of Buffalo, Buffalo, NY, USA, 2008.

[7] G. Chukkapalli, C. Guda, and S. Subramaniam, "SledgeHMMER: a web server for batch searching the Pfam database," *Nucleic Acids Research*, vol. 32, supplement 2, pp. W542–W544, 2004.

[8] "HMMER on the sun grid project," July 2009, http://www.psc.edu/general/software/packages/hmmer/.

[9] D. R. Horn, M. Houston, and P. Hanrahan, "ClawHMMER: a streaming HMMer-search implementation," in *Proceedings of the ACM/IEEE Supercomputing Conference, (SC '05)*, November 2005.

[10] GPU-HMMER, July 2009, http://www.mpihmmer.org/userguideGPUHMMER.htm.

[11] R. P. Maddimsetty, J. Buhler, R. D. Chamberlain, M. A. Franklin, and B. Harris, "Accelerator design for protein sequence HMM search," in *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*, pp. 288–296, New York, NY, USA, July 2006.

[12] "BLAST: Basic Local Alignment Search Tool," September 2009, http://blast.ncbi.nlm.nih.gov/.

[13] K. Benkrid, P. Velentzas, and S. Kasap, "A high performance reconfigurable core for motif searching using profile HMM," in *Procedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '08)*, pp. 285–292, Noordwijk, The Netherlands, June 2008.

[14] T. F. Oliver, B. Schmidt, Y. Jakop, and D. L. Maskell, "High speed biological sequence analysis with hidden Markov models on reconfigurable platforms," *IEEE Transactions on Information Technology in Biomedicine*, vol. 13, no. 5, pp. 740–746, 2009.

[15] J. P. Walters, X. Meng, V. Chaudhary et al., "MPI-HMMER-boost: distributed FPGA acceleration," *Journal of VLSI Signal Processing Systems*, vol. 48, no. 3, pp. 223–238, 2007.

[16] S. Derrien and P. Quinton, "Parallelizing HMMER for hardware acceleration on FPGAs," in *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP '07)*, pp. 10–17, Montreal, Canada, July 2007.

[17] L. Hunter, *Artificial Intelligence and Molecular Biology*, MIT Press, 1st edition, 1993.

[18] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[19] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[20] Y. Sun, P. Li, G. Gu et al., "HMMer acceleration using systolic array based reconfigurable architecture," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '09)*, p. 282, New York, NY, USA, May 2009.

[21] T. Takagi and T. Maruyama, "Accelerating hmmer search using FPGA," in *Proceedings of the19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 332–337, Prague Czech Republic, September 2009.

[22] R. B. Batista, A. Boukerche, and A. C. M. A. de Melo, "A parallel strategy for biological sequence alignment in restricted memory space," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 548–561, 2008.

[23] XtremeData Inc., July 2009, http://www.xtremedata.com/.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 646984, 10 pages
doi:10.1155/2012/646984

*Research Article*

# Optimizing Investment Strategies with the Reconfigurable Hardware Platform RIVYERA

**Christoph Starke, Vasco Grossmann, Lars Wienbrandt, Sven Koschnicke, John Carstens, and Manfred Schimmler**

*Department of Computer Science, Christian-Albrechts-University of Kiel, 24098 Kiel, Germany*

Correspondence should be addressed to Christoph Starke, chst@informatik.uni-kiel.de

Received 30 September 2011; Accepted 4 January 2012

Academic Editor: Thomas Steinke



The hardware structure of a processing element used for optimization of an investment strategy for financial markets is presented. It is shown how this processing element can be multiply implemented on the massively parallel FPGA-machine RIVYERA. This leads to a speedup of a factor of about 17,000 in comparison to one single high-performance PC, while saving more than 99% of the consumed energy. Furthermore, it is shown for a special security and different time periods that the optimized investment strategy delivers an outperformance between 2 and 14 percent in relation to a buy and hold strategy.

## 1. Introduction

The goal of technical financial market analysis is to predict the development of indices, stocks, funds, and other securities by evaluating the charts of the past. A method to find such predictions can lead to an investment strategy. Many well-known chart-analysis methods (e.g., Elliot waves [1], Bollinger Bands [2]) try to extract patterns from the charts, expecting that such patterns will come up in similar ways again in the future. There are more than 100 different chart-analysis methods but their success is doubted [3]. In most cases, the current development of the markets significantly affects the quality of the different investment strategies. Since the business volume per year on the worldwide stock markets is more than USD 35 trillions [4], it is not surprising that successful investment strategies are in the focus of intensive research.

In general, there are lots of indicators influencing the chart of a security. Those are not only economical and political indicators but also psychological ones. It is very difficult to decide which weight should be assigned to which indicator, the more so as there are known and unknown tradeoffs between different indicators. Furthermore, weights change in time. Recent papers [5–8] try to apply data mining methods on historical market rates, in order to find investment strategies that perform significantly above the average. This approach is extreme compute-intensive since every day there are millions of quotations that are fixed worldwide. Even with the use of high-performance computers the reduction of this amount of data is required. But as shown in the literature [5–8], data mining helps to keep the essential information contents in order to come to successful investment strategies.

In this paper, we present an investment strategy using a novel data mining method, which is discussed in Section 2. It results in a performance significantly above average for certain periods. It is based on the idea of an iterative search for an optimal set of indicator weights in the space of all possible weights of the indicators. Since this space grows exponentially with the number of indicators, the method is very compute-intensive but can optimally be parallelized. Therefore, an FPGA implementation seems to be very promising, because the computational core can be kept very simple and small in hardware. The two phases of the method have been implemented on the FPGA-based massively parallel computer RIVYERA. The RIVYERA architecture and its idea of efficiently exploiting 128 modern FPGAs in parallel are explained in Section 3.

Section 4 describes the architecture and the implementation of one processing element for the main computation. The speedup achieved by such an FPGA approach is investigated in Section 5. For different time intervals, the advantage in comparison to a single buy-and-hold strategy is determined. We do not want to discuss the investment strategy itself in this paper. Instead, the main focus here lies on the improvement in terms of speed, energy, and cost efficiency of the new method in comparison to an implementation on a sequential computer architecture. Section 6 summarizes the results and concludes the paper.

## 2. The Process of Optimizing an Investment Strategy for Securities

For a single security $P$ a successful strategy for buying and selling is desired. For simplicity, in this paper let $P$ be an investment fund that can be traded without trading costs (there are several discount brokers offering such funds, e.g., Vanguard in USA, InvestSMART Financial Services Pty Ltd in Australia, European Bank for Fund Services GmbH in Germany). Since the taxation regulations are varying in different countries, these are not considered here either.

We consider $n$ indicators $I_0, I_1, \ldots, I_{n-1}$ that might have influence on the chart of $P$. Typical indicators are S&P500, Nikkei225, EuroStoxx50, EUR/USD, and so forth. In other methods of technical analysis $P$ itself is used as an indicator as well. We consider a time interval of the past consisting of $m+1$ subsequent trading days $d_0, d_1, \ldots, d_m$. $m$ should be large enough to get significant results (e.g., $m \geq 125$).

$R$ is an $m \times n$ matrix, where $R_{i,j}$ is the percentage difference of the indicator $I_j$ from $d_{i-1}$ to $d_i$. The vector $R_i$ is the $i$th row of the matrix $R$: $R_i = (R_{i,0}, R_{i,1}, \ldots, R_{i,n-1})$. The required data for such a matrix can either be collected or downloaded from some trading platform in the internet.

At time $d_0$ we assume a cash capital of one million EUR and a depot with $D_0 = 0$ pieces of the security $P$. The value of one million has been chosen in order to be able to abstract from rounding errors. The results for other starting values can be computed proportionally. Generally, let $C_i$ be the cash money, $D_i$ the number of pieces of $P$ in the depot, and $Z_i$ the total property at day $d_i$ ($0 \leq i \leq m$). The fund considered here has exactly one market price $P_i$ per day. Therefore, the following condition holds:

$$Z_i = C_i + D_i \cdot P_i. \tag{1}$$

We are looking for a function $f(R_i)$ that computes the decision of buying or selling a certain amount of $P$ from the values of $R$ known up to $d_i$. The output of $f$ is on the one hand the decision either to buy, to do nothing, or to sell, and, on the other hand, the amount for the first and the last case. The optimal function of this kind is the one that maximizes the value of $Z_m$. This approach is motivated by the assumption of technical analysis that a successful strategy of the past will also be successful in the future.

In order to simplify the search for $f$, we consider only functions of the kind $f(R_i) = \sum w_j \cdot R_{i,j}$. The weight vector is denoted by $w = (w_0, w_1, w_2, \ldots, w_{n-1})$. We define $w^*$ as the vector which yields a maximum value for $Z_m$. A positive value of $f$ is a buy indication, a negative sell indication. The amount of $P$ to be traded is $Z_{i-1} \cdot |f(R_i)|$. A buy at day $d_i$ is limited to $C_{i-1}$ in order not to overdraw the cash account and a sell is limited to $D_{i-1}$, accordingly. The decision to cut down on functions of the kind $f(R_i) = \sum w_j \cdot R_{i,j}$ is based on the assumption that the influence of the different indicators is almost linear. Although this cannot be proven here, the results with this simplification are already remarkable. However, it is still worth to investigate modifications of this method with nonlinear functions.

There is one problem with investment funds: at that point in time at the day $d_i$ where the trading decision is made, the exact value of $P_i$ is not known. Therefore, at this point of time it is not possible to compute the exact number of pieces to be traded without overdrawing the cash account. For a buy order, we therefore transmit not the number of pieces but the amount of money for which we want to buy pieces. Vice versa, for a sell order, we should transmit the exact number of pieces to be sold.

Let $B_i$ be the amount of money for which pieces of $P$ should be bought at $d_i$ and $S_i$ the number of pieces to be sold at $d_i$. Obviously, the following condition holds: $(B_i = 0) \vee (S_i = 0)$.

Furthermore, if $f(R_i) \geq 0$, it holds:

$$\begin{aligned} B_i &= \min\left(Z_{i-1} \cdot f(R_i), C_{i-1}\right), \\ S_i &= 0, \\ C_i &= C_{i-1} - B_i, \\ D_i &= D_{i-1} + \frac{B_i}{P_i}. \end{aligned} \tag{2}$$

And if $f(R_i) < 0$,

$$\begin{aligned} B_i &= 0, \\ S_i &= \min\left(Z_{i-1} \cdot \frac{|f(R_i)|}{P_{i-1}}, D_{i-1}\right), \\ C_i &= C_{i-1} + S_i \cdot P_i, \\ D_i &= D_{i-1} - S_i. \end{aligned} \tag{3}$$

It is computationally unfeasible to determine $w^*$ with a brute force approach, even on a supercomputer. If one considers only 100 different values for each of 8 indicators, then there are $100^8$ different combinations of those. Using a calibration period of 26 weeks, with 5 trading days per week the required number of computations of the function $f$ would be

$$26 \cdot 5 \cdot 10^{16} = 1.3 \cdot 10^{18}. \tag{4}$$

As specified in the following examinations, even the presented RIVYERA implementation would require 377 days to evaluate that number of combinations. Instead of such a brute force method, we use an iterative approach: initially a very rough grid of 16 values per indicator is used. For each of the $16^8$ weight vectors, the final property $Z_m$ is computed. The areas in the grid, where $Z_m$ is relatively large, are the targets of the next iteration: in the environment of

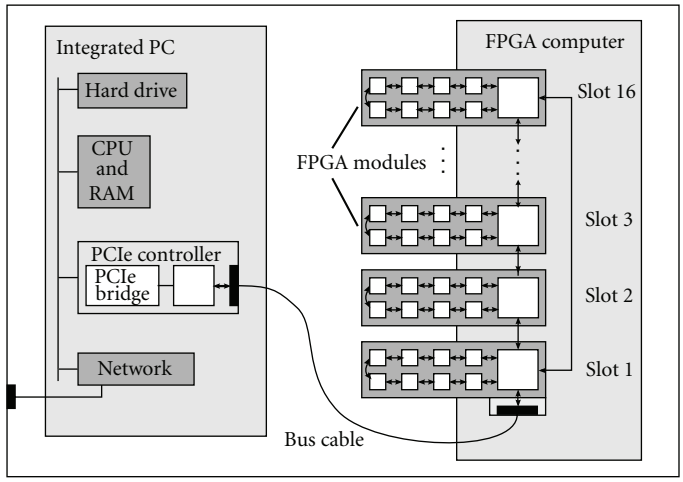



Figure 1: RIVYERA S3-5000 hardware structure.

the corresponding weights the grid, is refined. If a component of a promising weight vector is at the boundary of the grid then the grid is extended in this direction. In the same way, we keep on refining the grid. Already after 100 iteration steps, the results are satisfying. In each iteration step $16^8$ weight vectors are considered, resulting in

$$26 \cdot 5 \cdot 16^8 \approx 5 \cdot 10^{11} \tag{5}$$

calculations of the function $f$ plus the resulting calculation of the development of the depot value under the assumption that the corresponding buying and selling decisions are taken into account. The process of refining the grid is part of the host system and not specified in this paper.

The high computational effort is caused by the exhaustive search concerning the evaluation of $w^*$. This calculation can be remarkably accelerated by an FPGA-based implementation. Hence, the concrete task is the following: How can we accelerate the identification of the optimal weight vector $w^*$ which maximizes $Z_m$ out of a given set of weight vectors?

## 3. FPGA-Based Hardware Platform RIVYERA

Introduced in 2008, the massively parallel FPGA-based hardware platform RIVYERA [9] is the direct successor of the COPACOBANA, presented in 2006 for cost optimized breaking 56 bit DES ciphers in less than two weeks [10]. Besides applications in cryptanalysis (e.g., [11]), RIVYERA finds its applications in the fields of bioinformatics [12–14] and now stock market analysis, as described in this paper.

For the application presented here, the specific RIVYERA S3-5000 is used, distributed by SciEngines GmbH [15]. RIVYERA is designed to be a completely scalable system consisting of two basic elements. Firstly, the in-built multiple FPGA-based supercomputer provides the resources for parallel high-performance applications (Figure 1, right side). Secondly, a standard server grade mainboard equipped with an Intel Core i7-930 processor, 12 GB of RAM, and 2 TB of hard disk space, provides the resources for quick pre- and postprocessing purposes (Figure 1, left side). The RIVYERA S3-5000 is powered by two 650 W supplies and packed in a standard rack mountable 3 U housing. It is running a standard Linux operating system and, therefore, presents an independent system. The details are discussed briefly in the following.

The FPGA-based supercomputer consists of a backplane and up to 16 FPGA cards (fully equipped for the application described in this paper). Each FPGA card is equipped with eight user configurable Xilinx Spartan3-5000 type FPGAs and one additional FPGA as communication controller. In total, these are 128 user configurable FPGAs. Additionally, a DRAM module with a capacity of 32 MB is directly attached to each user FPGA.

All FPGAs are connected by a systolic-like bus system. Each FPGA on an FPGA card is connected with two neighbors forming a ring including the communication controller. The FPGA card slots are connected to each neighboring slot as well on the backplane, providing the connections between the communication controllers on each FPGA card. The communication is physically realized by high-throughput symmetric LVDS point-to-point connections. The communication of the FPGA-based computer to the host-mainboard follows a connection via PCIe controller card directly to a communication controller on a chosen FPGA card. For applications requiring a higher bandwidth from the host system to the FPGA-based computer, more than one PCIe controller may be attached to other FPGA cards as well. For a configuration as used for this application, the measured net bandwidth from the host to the FPGA computer reaches up to 66 MB/s. Of course, the latency will be different dependent on which clients are communicating with each other, according to the length of the communication chain.

For application development, the RIVYERA provides an API for each of the two basic elements, that is, an API controlling the data transfer between the host software and the FPGAs including broadcast facilities, and an API for the user defined hardware configuration of the FPGAs

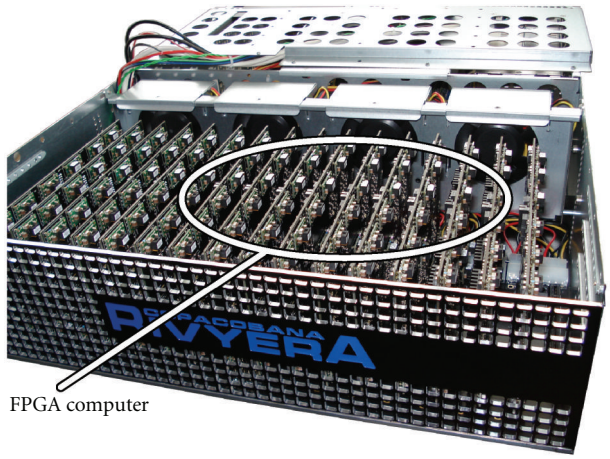



Figure 2: RIVYERA S3-5000. The 16 FPGA-cards forming the FPGA-computer are highlighted. The integrated standard PC cannot be seen behind the cover.

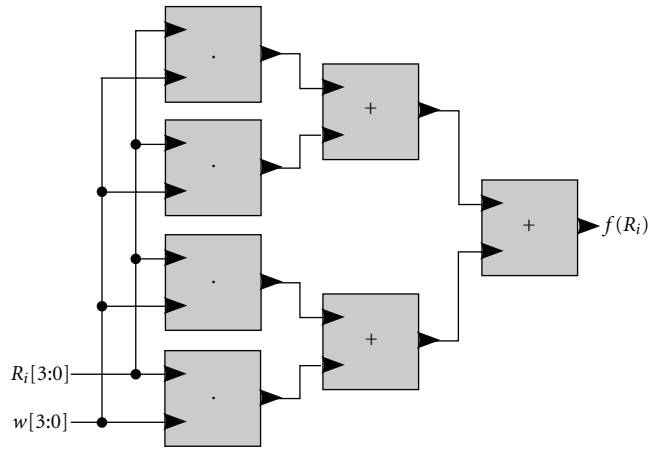



Figure 3: Calculation of $f(R_i)$ with four indicators.

controlling the data transfer to other FPGAs and the host as well.

A picture of the RIVYERA S3-5000 is shown in Figure 2.

## 4. Processor Architecture

The FPGA-based part of the presented algorithm is based on exhaustive searches. As different weight vectors can be evaluated independently, the algorithm is suitable for massive parallelization. Therefore, the following description of the technical implementation only considers a single FPGA. Assuming uniform programming of all available FPGAs and an equally divided search space, the computational speed rises approximately linear with the number of FPGAs. According to the RIVYERA platform, the implementation presented here is optimized for Xilinx Spartan3-5000 FPGAs [9, 16].

The key aspect concerning the identification of valuable weight vectors is the calculation of the score $Z_m$ for every possible element of the search space. Since these evaluations are the fundamental issue of the computational effort, the success of creating an efficient processor architecture is directly linked to the performance of the underlying implementation of the scoring function. Thus, the main objective, and therefore starting point for the design of the processing element, should be the creation of a scoring unit with a high throughput.

*4.1. Scoring Pipeline.* The evaluation of $Z_m$ consists of repetitive computations of the sequences $C_i$ and $D_i$. Therefore, the throughput of the scoring unit is directly connected to the performance of the computation of these two sequences. Thus, despite the high spatial cost, the advantages of a pipeline architecture are persuasive. The implementation presented here is based on pipelines that yield a new pair $(C_i, D_i)$ in every clock cycle. As the values $C_i$ and $D_i$ are defined recursively, the pipeline has to wait for its own outputs. Thus, to avoid idle time, $l$ scores for different weight vectors are evaluated concurrently, where $l$ is the length of the longest cyclic path. Hence, $l$ is given by the number of clock cylces that are necessary to compute $C_i$ and $D_i$ from $C_{i-1}$ and $D_{i-1}$.

Basically, the structure can be subdivided into three segments. The first one is described by the function $f(R_i) = \sum w_j \cdot R_{i,j}$. Assuming $n$ indicators, the calculation of $f(R_i)$ needs $n$ multiplications and $n-1$ additions. The corresponding structure for $n = 4$ is shown in Figure 3. As all following calculations directly depend on $f(R_i)$, this computation is part of the longest path of the pipeline. Hence, the additions should be arranged in a way that only the minimum number of steps (1 multiplication and $\lceil \log_2 n \rceil$ additions) is required. However, the path is not element of the longest cyclic path because $w_j$ and $R_{i,j}$ do not depend on the outputs of the pipeline. A different arrangement of the additions has no effect to $l$.

Due to resource reduction the buy order size $B_i$ and the sell order size $S_i$ are combined to a general order size $O_i$. A negative value indicates a sell order, and a positive value denotes a buy order. Instead of the sequence $D_i$, the pipeline calculates the values $D'_i = D_i \cdot P_i$ as it enables to the evaluate $Z_m$ just by one addition. The calculation of $O_i$ is given by the following instruction:

$$O_i := \begin{cases} -D'_i, & \text{if } O^*_i \le -D'_i, \\ C_i, & \text{if } C_i \le O^*_i, \\ O^*_i, & \text{else.} \end{cases} \tag{6}$$

When the evaluation of $f(R_i)$ is finished, the order size $O_i$ at day $i$ is computed as shown in Figure 4. The intermediate result $O^*_i = (D'_i + C_i) \cdot f(R_i)$ is restricted to $O_i$ by the usage of two multiplexers and corresponding comparators. The total property $C_i + D'_i$ and the negative depot value $-D'_i$ do not depend on $f(R_i)$ and, thus, can be calculated in parallel to its evaluation. Hence, the longest path of the pipeline is extended by the multiplication and the comparator chain.

After the calculation of the order size, new cash and depot values are computed. The value $i$ that identifies the day of

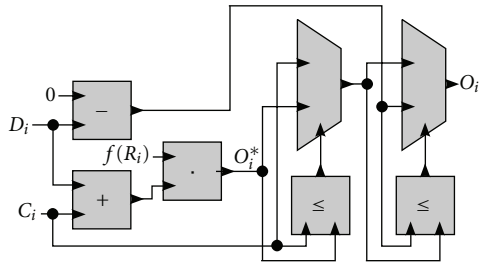



Figure 4: Calculation of order size $O_i$.

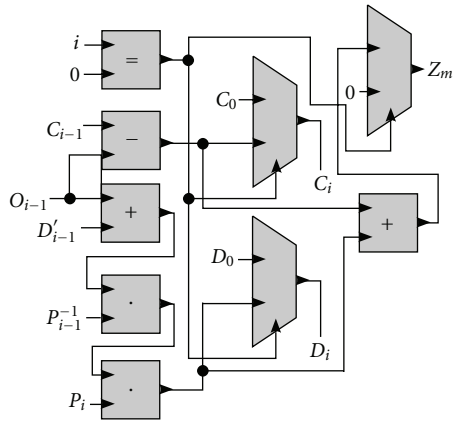



Figure 5: Evaluation of $Z_m$.

the historical data set rises by 1 since the end of the given time period ($i = m$) is reached. In this case $i$ is set to 0 which implies the start of the evaluation of a new weight vector. The sequences $C_i$ and $D'_i$ are reset to the default values $C_0$ and $D'_0$. In the same clock cycle the sum of $C_m$ and $D'_m$ is calculated and transmitted to the multiplexer that refers to $Z_m$:

$$(C_i, D'_i) := \begin{cases} (C_0, D'_0) & \text{if } i = 0, \\ \left(C_{i-1} - O_{i-1}, (D'_{i-1} + O_{i-1}) \cdot \dfrac{P_i}{P_{i-1}}\right) & \text{else.} \end{cases} \quad (7)$$

In comparison to a multiplication, a division is much more expensive in regard to resource usage [16]. As a consequence, the quotient $P_i/P_{i-1}$ is realized as the multiplication $P_i \cdot P_{i-1}^{-1}$. On the one hand, this implies the additional calculation and storage of inverse elements. On the other hand, every calculation needs to be done only once and can be outsourced to the host system. Likewise, the additive memory usage can be disregarded as we will see in Section 4.3.

As considered, the algorithm is trivially parallelizable. The computational speed depends linearly on the number of FPGAs. Likewise, this statement can be assigned on the number of pipelines. But how many pipelines can be synthesized on an FPGA and are there further possibilities to increase that number?

Table 1: Synthesis result with floating point representation.

| Indicators | Slices | Multipliers |
|---|---|---|
| 8 | 8584 (25%) | 32 (30%) |
| 16 | 14064 (42%) | 44 (42%) |

*4.2. Optimized Fixpoint Representation.* All in all, one scoring pipeline is built of $n + 3$ multiplications, $n + 2$ additions, 2 subtractions, 3 comparators, and 5 multiplexers where $n$ is the number of indicators (see Figure 6) that are $15 + 2n$ operations. A Spartan 3–5000 FPGA consists of 8,320 Configurable Logic Blocks which can be separated in 33,280 Slices [16]. Additionally, 108 dedicated $18 \times 18$ bit multipliers can be assigned for synthesis.

The allocation report of two synthesis results is shown in Table 1. A single precision floating point representation of all variables is assumed in both cases. Using 32 multipliers, 8 indicators yield a consumption of 25% of the available slices. Assuming that 10% of the slices are reserved for further control units, three pipelines can be synthesized on the FPGA. In case of 16 indicators, additional 17% are required. The drastic increase results from the 8 additional adders and multiplicators and the comparatively high spatial cost of floating point units [16]. An important point is the synchronization of the different pipeline stages. For example, the third stage (see Figure 5) receives, amongst others, the input values $O_{i-1}$ and $P_i$. While $P_i$ is given, $O_i$ is only known after several calculations. Hence, to provide synchronicity, the transfer of $P_i$ is delayed using shifting registers. The longest cyclic path consists of 2 additions, 3 multipliers, 2 comparators, and 3 multiplexers. As an extension of the pipeline implies the requirement of more shifting registers, the path should be as short as possible. Optimized in terms of space, the longest cyclic path comprises $l = 57$ clock cycles. All in all, only two pipelines are possible in this case.

To counter that problem, a fixpoint representation will be introduced in the following. The idea is motivated by the fact that many of the given values are located in limited ranges. For example, the daily price fluctuations in $R$ rarely exceed the interval $[-10\%, 10\%]$. That is the reason why the values of $R$ will be stored in 18 bits where the decimal place is coded in 12 bits and the new codomain is the interval $[-32\%, 32\%)$ with a precision of $2^{-12}\%$. Likewise, the elements of the weight vector will be stored in 18 bits. While a decimal place of 12 bits seems to be the best tradeoff between overflow immunity on the one hand and precision on the other hand, the range of the weight vectors may be determined specifically for every use case. Cash, depot value, and stock prices are stored in integer values in cent. The inverse prices are multiplied with $2^{32}$ and also stored in integer values.

The 18-bit representation of $R_{i,j}$ and $w_j$ promotes the efficient usage of the dedicated $18 \times 18$ multipliers. Furthermore, the transfer from floating point to fixpoint units leads to a considerable decrease of the allocated resources.

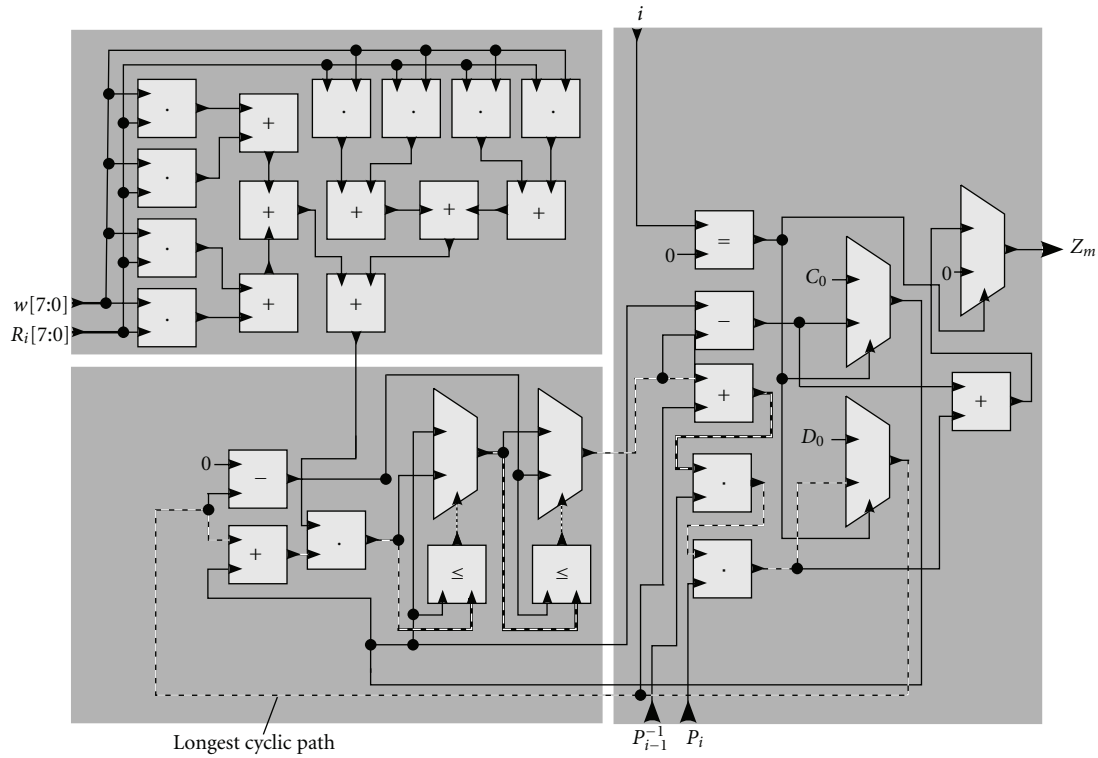



Figure 6: Scoring pipeline for 8 indicators.

The length of the longest cyclic path can be reduced to 37. As shown in Table 2, the available resources suffice for up to 6 pipelines per FPGA.

*4.3. FPGA Overview.* The pipelines are triggered synchronously. The trading period $d$ and the corresponding historical data are set globally for all scoring units. Since $l$ independent score evaluations are calculated in parallel in every pipeline, the value of $d$ has to change only once every $l$ clock cycles. To trigger the pipeline in the $i-1$th recursion, the historical information of day $i$ is necessary. This set consists of the vector of price fluctuations $R_i$ and the values $P_i$ and $P_{i-1}^{-1}$. To transfer these values within one clock cycle, the historical data of day $i$ $H_i$ is stored in a single Block RAM word. Such a word $H_i = (P_{i-1}^{-1}, P_i, R_i)$ consists of $32+32+18\cdot n$ bits, for example, 208 bits for $n = 8$ indicators. Spartan3-5000 provides 104 RAM blocks with 1,872 KB in total [16]. This is obviously enough in our case, as it suffices for over 9000 days relating to 8 indicators.

As the optimization is based on an exhaustive search, it is necessary to determine the search space. The declared objective is to identify the optimal weight combination for 8 indicators. 8 possible candidates are given for every indicator. So, the search space is declared by an $8 \times 8$ matrix. Every row describes an indicator and consist of 8 values. Each of these values can be used as a weight to the correspondent indicator. As there are 8 possible candidates for each of 8 indicators, the number of possible weight vectors is $|W| = 8^8 \approx 16.7$ million. So, one FPGA is able to calculate the optimal weight vector out of 16.7 million combinations. One unique combination for every pipeline has to be calculated in every clock cycle. To accomplish this, every possible combination is declared by an 24-bit identifier in the range of $[0, |W|-1]$. An equally divided subspace is $([0, |W|/6 - 1], [|W|/6, |W|/3 - 1], \ldots]$ assigned to every pipeline. The weight vector is extracted by masking the identifier. The bits $3\cdot j+2$ to $3\cdot j$ show the position of coefficient $w_j$ of the weight vector $w$. For example, the identifier $387_{10} = 110.000.011_2$ references the matrix items $W[0][3]$ $(011_2 = 3_{10})$ for $w_0$ and $W[2][6]$ for $w_2$. This interpretation is very efficient as the effort of bit masking is comparatively small. Thus, 6 different weight vectors can be selected in a single clock cycle and assigned to the pipelines.

Table 2: Synthesis result with fixpoint representation.

| Indicators | Slices | Multipliers |
|---|---|---|
| 8 | 4801 (14%) | 12 (11%) |
| 16 | 5395 (16%) | 20 (19%) |

As the data flow is synchronous, the scores $Z_m$ of all pipelines are calculated at the same time. Assuming 6 pipelines, 6 results are returned per clock cycle. Obviously, it is neither possible nor does it make sense to store $8^8$ values. Likewise, the effort to administrate a list of the best scores is too high as it implies the sorting of 6 results into the list in a single clock cycle. The examination of this problem shows that a good tradeoff is the storage of the best result of every pipeline. Utilizing 6 pipelines and 128 FPGAs, 768

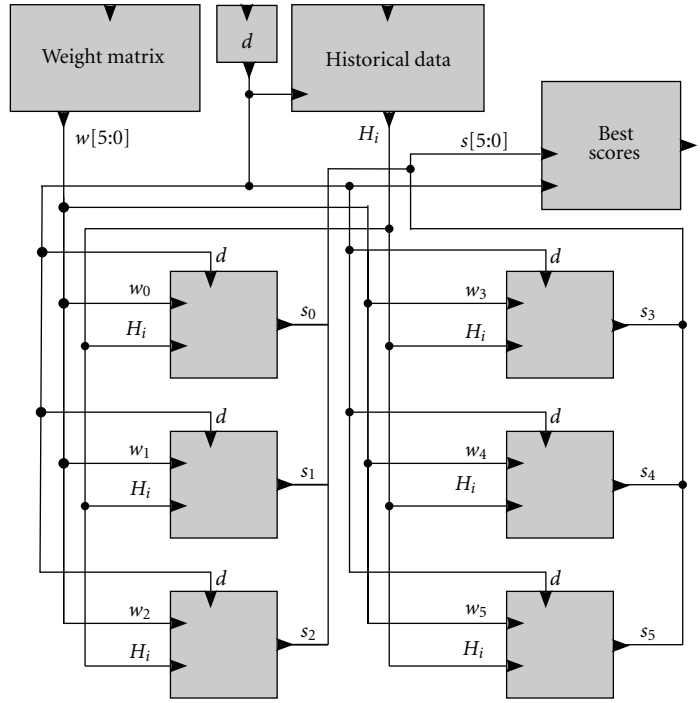



Figure 7: Outline of the processor architecture.

results are evaluated in every iteration. This set seems to be widespread enough to calculate new weight coefficients for the next iteration. An overview of the FPGA structure is shown in Figure 7.

## 5. Results and Performance Analysis

For further research, we will now consider results and performance for a certain security, the investment fund DWS Convertibles, ISIN DE0008474263, that is operating internationally.

As described in Section 2 (referred as calibration phase in the following), the optimal weight vector $w^*$ is determined for the security and furthermore for a randomly chosen time period of 26 weeks (calibration time interval). We chose 8 indicators that widely represent the current economical environment: S&P 500, DAX, EuroStoxx 50, ASX 200, Nikkei 225, Hang Seng, S&P 500 Future, and EUR/USD. The goal is to find $w^*$ with the maximal value of $Z_m$.

The computational effort with 8 indicators is already rather high. In this paper, we disclaim to investigate more indicators since, on the one hand, these 8 indicators represent the activities on the international stock markets to some high degree, and on the other hand, the results with this restriction are already remarkable.

We now focus on the investment strategy where at day $d_i$ for indicators $I_j$ the $R_{i,j}$ are calculated and then from $f(R_i)$ the volume of buying or selling orders is computed based on the value of $w^*$. To determine the quality of the vector $w^*$, we test it in a different period of time referred to as the evaluation phase. Of course, this makes only sense for a time interval (the so-called evaluation time interval) which does not overlap with the calibration time interval. We have chosen three different evaluation time intervals of 26 weeks as well. The question is whether or not the new investment strategy gives an outperformance in comparison to a buy-and-hold strategy. Buy-and-hold means $P$ is bought at the beginning of the evaluation time interval and sold at the end.

Figure 8 shows an example of the chart of the security in comparison to the performance of our investment strategy with the same security within three different evaluation time intervals $T_k$ of 26 weeks each, $k \in \{1, 2, 3\}$. $T_1$ (2009-09-14–2010-03-15) is a period where tendency for the fund is rising. $T_2$ (2010-09-27–2011-03-28) is a period without a clear tendency and $T_3$ (2011-03-28–2011-09-26) is a period where tendency for the fund is falling.

The values of $w_k^*$ had been determined for each $T_k$ in the iterative way described previousuly. The resulting investment strategy $S_k$ was then applied for the evaluation time intervals $T_e$, where $e \in \{1, 2, 3\}$ and $e \neq k$. The chart $P_{k,e}$ shows the performance of the monetary assets in the evaluation time interval $T_e$ using investment strategy $S_k$.

In all time intervals, an outperformance of the investment strategies $S_k$ over $P$ between 2% (see $P_{2,1}$ in Figure 8) and 14% (see $P_{1,3}$ in Figure 8) can be seen. Although this is no proof in a mathematical sense that such an investment strategy can be applied to arbitrary securities in arbitrary time periods, it seems to be very promising to further improve the method described here.

Considering computing performance as well, the RIVYERA or similar computer architectures are perfectly suited for such research. Table 3 shows a comparison between the RIVYERA-based approach and a PC version of the algorithm implemented in C. The test system uses an Intel Core i7–970

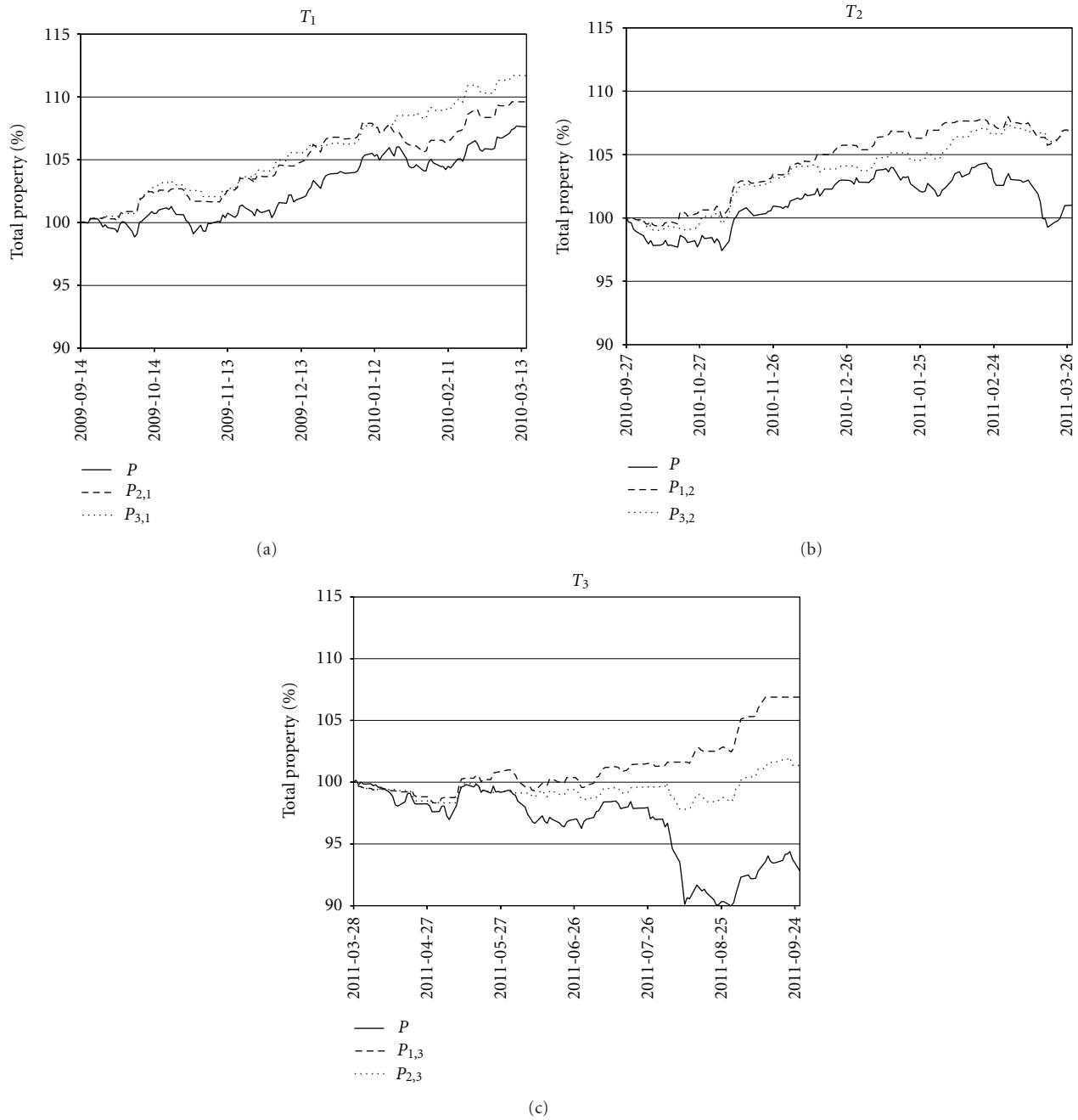



Figure 8: Performance of the investment strategy in different evaluation intervals.

with $6 \times 3200$ MHz, an ASRock X58 Extreme mainboard and 8 GB GeIL DIMM DDR3-1066 RAM. The implementation uses all cores of the processor. In addition, the improved number representation is used in the PC version as well.

$128 \cdot p$ pairs $(C_i, D_i')$ are calculated in every clock cycle on RIVYERA where $p$ is the number of pipelines per FPGA. The clock rate of the implementation is 50 MHz. Assuming 8 indicators, 6 pipelines can be synthesized on an FPGA. This yields $128 \cdot 6 \cdot 50,000,000 = 38.4$ billion pairs per second. Examinations of the PC version denote that 2.26 million calculations per second are possible on the specified test system. The conclusion is a speedup of about 17,000. While RIVYERA requires up to 1300 W, 300 W is supposed for a standard PC. Accordingly, the power consumption is reduced by up to 99.975%.

Of course, such a comparison yields a number of questions. Intel declares 76.8 GFLOPs for i7–970 [17]. The presented FPGA design needs $15+2n = 31$ operations for $n = 8$ indicators. Assuming that the PC version manages to work with the same number of operations, one could deduce that the referred processor reaches up to $78,600,000,000/31 = 2.54$ billion pairs per second. This is more than 1,000 times

Table 3: Comparison of PC and RIVYERA.

| Runtime (calibration phase: 26 weeks) | | |
|---|---|---|
| Number of weight vectors | PC | RIVYERA |
| 1 billion | 44 h 56 m | 9.15 s |
| 50 billion | 93 d 14 h | 7 m 37 s |
| 1 trillion | 5.13 years | 2 h 32 m |
| Power consumption | | |
| Number of weight vectors | PC (300 W) | RIVYERA (1300 W) |
| 1 billion | 13.48 KWh (2,70€) | 3.31 Wh (0,0006€) |
| 50 billion | 0.674 MWh (137,49€) | 165.20 Wh (0,04€) |
| 1 trillion | 13.48 MWh (2695,80€) | 3.31 KWh (0,66€) |

faster than the actual implementation. So, what is the reason for this gap?

In fact, the computing power of the processor is not the bottleneck. The main problem is located in the intensive memory communication. Even a cache-optimized version needs several RAM accesses (and of course many cache accesses) to calculate a single pair. The pipeline structure cannot directly be translated but only be simulated by further memory instructions. In contrast, the FPGA Block RAM modules are triggered in parallel to the actual calculations. Thus, there is absolutely no latency concerning memory operations. This is a reason why this algorithm is very suitable for massively parallel computing. A further interesting issue would be a comparison in performance using GPGPU.

As well, the time complexity of the presented algorithm differs in regard to the different platforms. On standard processors, the complexity is $\mathcal{O}(w_{\max}^n \cdot n \cdot m)$ where $w_{\max}$ denotes the maximum number of possible coefficients for one indicator. The factor $n$ occurs because the evaluation of $f(R_i)$ needs $n$ multiplications and $n-1$ additions that has to be executed sequentially. A RIVYERA pipeline calculates one pair $(C_i, D_i')$ per clock cycle in every case. So, there is obviously no such dependency. Therefore, the time complexity is $\mathcal{O}(w_{\max}^n \cdot m)$. However, the dependency on $n$ is not erased by this approach. While the size of a standard processor remains constant for an increasing $n$, more adders and multipliers are necessary in terms of an FPGA-based implementation. According to this, the spatial complexity is $\mathcal{O}(n)$. As this may lead to less pipelines per FPGA, an indirect influence to the runtime cannot be concealed.

## 6. Conclusion

The FPGA-machine RIVYERA is very suitable for optimization of the investment strategy as it was presented in this paper. A speedup of 17,000 and an energy saving of more than 99% in comparison to one single high-performance PC has been determined. The investment strategy which is optimized with RIVYERA delivers for the special investment fund and different time periods reviewed a significant outperformance in relation to a buy and hold strategy.

Several other securities for different time periods were tested. Although always the same, simple indicators were used, the optimization of the investment strategy by using RIVYERA delivered almost in every case a significant outperformance.

## References


[1] R. N. Elliott, *The Wave Principle*, 1938.

[2] J. Bollinger, *Bollinger on Bollinger Bands*, McGraw-Hill, 2001.

[3] C. Park and S. Irwin, "What do we Know about the profitability of technical analysis?" *Journal of Economic Surveys*, vol. 21, pp. 786–826, 2007.

[4] M. Nagler, *Finanzcrash und Systemkrise*, 2008.

[5] M. Gavrilov, D. Anguelov, P. Indyk, and R. Motwani, "Mining the stock market: which measure is best?" in *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2000.

[6] K. S. Kannan, P. S. Sekar, M. M. Sathik, and P. Arumugam, "Financial stock market forecast using data mining techniques," in *Proceedings of the International MultiConference of Engineers and Computer Scientists (IMECS '10)*, vol. 1, pp. 555–559, 2010.

[7] S. Langdell, "Examples of the use of data mining in financial applications," in *Financial Engineering News*, p. 2002.

[8] T. Rathburn, *Data Mining the Financial Markets, Part 1*, Data Mining, Down Under Forums, 2007.

[9] G. Pfeiffer, S. Baumgart, J. Schröder, and M. Schimmler, "A massively parallel architecture for bioinformatics," in *Proceedings of the 9th International Conference on Computational Science (ICCS '09)*, vol. 5544 of *Lecture Notes in Computer Science*, pp. 994–1003, Springer, 2009.

[10] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, A. Rupp, and M. Schimmler, "How to Break DES for €8,980," in *Proceedings of the Workshop on Special-Purpose Hardware for Attacking Cryptographic Systems*, Cologne, Germany, 2006.

[11] J. Fan, D. V. Bailey, L. Batina, T. Guneysu, C. Paar, and I. Verbauwhede, "Breaking elliptic curve cryptosystems using reconfigurable hardware," in *IEEE Field Programmable Logic*, pp. 133–138, 2010.

[12] M. Schimmler, L. Wienbrandt, T. Guneysu, and J. Bissel, "COPACOBANA: a massively parallel FPGA-based computer architecture," in *Bioinformatics High Performance Parallel Computer Architectures*, B. Schmidt, Ed., pp. 223–262, CRC Press, 2010.

[13] L. Wienbrandt, S. Baumgart, J. Bissel, F. Schatz, and M. Schimmler, "Massively parallel FPGA-based implementation of BLASTp with the two-hit method," *Procedia Computer Science*, vol. 4, International Conference on Computational Science (ICCS '10), pp. 1967–1976, 2011.

[14] L. Wienbrandt, S. Baumgart, J. Bissel, C. M.Y. Yeo, and M. Schimmler, "Using the reconfigurable massively parallel architecture COPACOBANA 5000 for applications in bioinformatics," *Procedia Computer Science*, vol. 1, International Conference on Computational Science (ICCS '10), no. 1, pp. 1027–1034, 2010.
[15] SciEngines GmbH: http://www.sciengines.com/.
[16] Xilinx Inc.: Xilinx UG331 Spartan-3 Generation FPGA User Guide, March 2011, http://www.xilinx.com/.
[17] Intel Corporation: Intel Core i7-900 Desktop Processor Series, http://www.intel.com/.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 241439, 10 pages
doi:10.1155/2012/241439

*Research Article*

# The "Chimera": An Off-The-Shelf CPU/GPGPU/FPGA Hybrid Computing Platform

**Ra Inta, David J. Bowman, and Susan M. Scott**

*The Centre for Gravitational Physics, Department of Quantum Science, The Australian National University, Canberra, ACT 0200, Australia*

Correspondence should be addressed to Ra Inta, ra.inta@anu.edu.au

Received 2 October 2011; Revised 4 January 2012; Accepted 4 January 2012

Academic Editor: Thomas Steinke



The nature of modern astronomy means that a number of interesting problems exhibit a substantial computational bound and this situation is gradually worsening. Scientists, increasingly fighting for valuable resources on conventional high-performance computing (HPC) facilities—often with a limited customizable user environment—are increasingly looking to hardware acceleration solutions. We describe here a heterogeneous CPU/GPGPU/FPGA desktop computing system (the "Chimera"), built with commercial-off-the-shelf components. We show that this platform may be a viable alternative solution to many common computationally bound problems found in astronomy, however, not without significant challenges. The most significant bottleneck in pipelines involving real data is most likely to be the interconnect (in this case the PCI Express bus residing on the CPU motherboard). Finally, we speculate on the merits of our Chimera system on the entire landscape of parallel computing, through the analysis of representative problems from UC Berkeley's "Thirteen Dwarves."

## 1. Computationally Bound Problems in Astronomical Data Analysis

Many of the great discoveries in astronomy from the last two decades resulted directly from breakthroughs in the processing of data from observatories. For example, the revelation that the Universe is expanding relied directly upon a newly automated supernova detection pipeline [1], and similar cases apply to the homogeneity of the microwave background [2] and strong evidence for the existence of dark matter and dark energy [3]. Most of these discoveries had a significant computational bound and would not have been possible without a breakthrough in data analysis techniques and/or technology. One is led to wonder the astounding discoveries that could be made without such a computational bound.

Many observatories currently have "underanalyzed" datasets that await reduction but languish with a prohibitive computational bound. One solution to this issue is to make use of distributed computing, that is, the idle CPUs of networked participants, such as the SETI@HOME project [4]. It is clear that a number of common data analysis techniques are common across disciplines. For example, LIGO's Einstein@HOME distributed computing project, designed to search gravitational wave data for spinning neutron stars, recently discovered three very unusual binary pulsar systems in Arecibo radio telescope data [5].

These are far from the only "underanalyzed" datasets from existing observatories, and this situation is expected to only compound as we look forward to an ever increasing deluge of data. For example, the Square Kilometer Array is expected to produce about an exa-byte *a day* [6] (to put this into perspective, it is estimated that *all* the stored information created by humanity is roughly 300 EB [7]); just for fairly basic operation, this alone will require to be close to the projected computing power of the world's single most powerful computing system [8] at the expected 2020 date of "first light." There are a number of robotic survey telescopes either already, or scheduled soon to be, on-line to detect transient events, from near-Earth objects such as asteroids [9] to distant GRBs [10].

It should be obvious that many other sectors have exponentially growing appetites for computation, from military [11] through financial [12], even cinematic applications require the most powerful HPC systems [13]. Considering

the common computational requirements of these systems, it is clear that a revolution in HPC technology is required in order to keep pace with projected needs.

## 2. Problems with Conventional Cluster-Based HPC Systems

Until very recently, the most powerful HPC systems (the "Top 500") were purely CPU based [8], although there is a very recent but significant shift towards the use of general-purpose graphical processor unit (GPU) coprocessing. However, many critics point out that the "Top 500" may not be representative of the true compute power of a cluster [14]. The negative corollary of the efficient compliance of CPUs with LINPACK is that the resulting rigid instruction set can compromise performance when performing operations not heavily dependent on dense linear algebra. As we show in Section 6 below, this is, only one feature of many that are vital to problems involving parallel computation.

Because power consumption, and hence heat generation, is proportional to clock speed, processors have begun to hit the so-called "speed wall" (e.g., Intel cancelled their 4 GHz processor line because of heat dissipation issues [15]). Furthermore, there is a growing awareness of the monetary cost of powering traditional HPC systems: over half the lifetime cost of a modern supercomputer is spent on electrical power [16]. Indeed, many computing clusters are sited near large generation plants in order to save on power transport costs [17].

It is also easy to forget that the concept of a "general purpose" microprocessor is a relatively recent idea—for example, only since the introduction of the 486 processor have Intel not used a dedicated floating point coprocessor. Meanwhile, hardware accelerators have occupied specialized niches, such as video processing and high-speed DSP applications. The most commonly available of these accelerators are the (general-purpose) graphical processor unit (GPU) and the FPGA. Easing or circumventing many of the issues with CPU-based HPC is becoming an attractive prospect to a growing cadre of consumers. Currently promising teraflop/s performance with a low initial capital outlay and without need of a specialized power supply or cooling facility, both the GPU and the FPGA are viable alternatives to purchasing many CPU-based units. Until recently, the development time associated with these platforms was considered rather high, especially for the FPGA. However, the programming interfaces for both have become more user-friendly. Finally, the future appears bright for both accelerator classes, as both have performance growth well exceeding Moore's Law, and consequently, that of CPUs [18].

## 3. Comparisons between CPUs and Hardware Accelerators

The long-standing workhorse of the vast majority of data analysts is the general-purpose CPU. Because it is expected to perform a range of different tasks, CPU processor designs cannot afford to specialize. Although the processor clock speeds are fairly high, it can take many cycles to perform an intensive computation, because it will be scanning for interrupts, and so forth. CPUs are generally very good at performing a multitude of separate tasks at a moderate speed and are efficient at moderating/coordinating a range of slave devices. The performance and merits of these devices should be fairly familiar to the reader.

Because GPU platforms were designed to efficiently perform linear operations on vectors and matrices, a general rule of thumb is that any operations that require intensive linear calculations are best made on a GPU. Many embarrassingly parallel computations rely on linear algebraic operations that are a perfect match for a GPU. This, in addition to the amount of high level support, such as C for CUDA, means they have become adopted as the hardware accelerator of choice by many data analysts. For example, a comparison of the GPU-based CUDA-FFT against the CPU-based FFTW3 on gravitational wave data analysis showed a 4X speedup for one million, and 8X for four million, points; this exact approach can also be used to detect radio transients with synthetic aperture array radio telescopes [19]. An excellent analysis of a range of algorithms heavily used in astronomy with applications including imaging, gravitational lensing, and pulsar characterization, implemented on GPUs, is given in [20].

FPGAs, on the other hand, of course, represent an entirely different approach to computing altogether. Because of their unrivalled computing flexibility, it can be difficult for the data analyst, used to a rigid instruction set-based processor, to be entirely comfortable with the low level required to construct an analysis pipeline. The majority of the applications in astronomy have been in instrumentation and data capture, such as the FPGA-based digital cross-correlator for synthetic aperture array radio telescopes [21, 22] or spectrometers [23]. However, there is a small but growing base of analysts willing to adopt an FPGA-based hardware acceleration solution, with applications including detection of gamma-ray pulsars [24]. There are a number of FPGA-based HPC facilities such as Janus [25] and Maxwell [26], and companies such as Starbridge, Inc. and Pico Computing, Inc. [11] offering FPGA computing solutions. A good survey of the state of FPGA-based HPC is given in [27].

Determining the relative strengths of each hardware acceleration class is highly algorithm (and data-type) dependent. There are many comparisons in the literature between FPGA, GPU and CPU, implementations of the same algorithms, ranging from random number generation [28] (where at 260 Gsample/s, FPGAs were found to be faster by a factor of 15 and 60 over a contemporaneous GPU and CPU resp.), video processing [29] (where FPGAs may have a significant advantage over GPUs when multiplying arrays of rank four or higher), convolution [30] (FPGAs are advantageous because of their pipelining and streaming abilities) to MapReduce [26, 31] (where the former show that GPUs considerably out-perform FPGAs and the latter show that an FPGA implementation of Quasi-Random Monte Carlo outperforms a CPU version by two orders of magnitude and beats a contemporaneous GPU by a factor of three), and least squares applications [32] (an FPGA implementation is

slightly worse than that for a GPU, which is in turn slightly worse than a CPU, for large matrices). Some more general overviews are given in [33, 34]. However, one must be careful not to generalize performance without consideration of the detailed technical specifications of the components. Both FPGA and GPU platform designs are in a state of unprecedented flux, and hence relative performance benchmarks are likely to change also. This caveat notwithstanding there are a number of distinctions amongst each hardware platform intrinsic to the underlying design features.

With the above considerations in mind, we present here a system that attempts to exploit the innate advantages of all three hardware platforms, in order to attack problems with a computational bound that would benefit from a mixed hardware subsystem "heterogeneous" approach.

## 4. The "Chimera" Heterogeneous CPU/GPU/FPGA Computing Platform

We originally conceived a platform that would exploit the advantages of both FPGA and GPU accelerations *via* a high-speed backplane interconnect system [35]. A schematic is shown in Figure 1.

There are a number of platforms that implement a heterogeneous FPGA/GPU/CPU system. The "Quadro-Plex Cluster" [36] is a sophisticated sixteen node cluster with two 2.4 GHz AMD Opteron CPUs, four nVidia Quadro FX5600 GPUs, and one Nallatech H101-PCIX FPGA in each node, with a thread management design matching that of the GPUs. However, it does not yet appear to implement a combination of FPGAs *and* GPGPUs within an algorithmic pipeline, which is essential for our applications. Also, the FPGA architecture was designed to mirror that of the microprocessor, including full double precision support, which in general will not optimize the potential performance of the FPGA. The "Axel" [37] system is a configuration of sixteen nodes in a Nonuniform Node Uniform System (NNUS) cluster, each node comprising an AMD Phenom Quad-core CPU, an nVidia Tesla C1060, and a Xilinx Virtex-5 LX330 FPGA. From the perspective of Axel, our proposed platform would conform to the Uniform Node Nonuniform System (UNNS) configuration, or perhaps an optimized version of each node within an Axel-type cluster. There is a "desktop supercomputer" comprising a single CPU, GPU, and FPGA [38], used to model coupled resonator optical waveguides, although unfortunately the architecture and configuration of this system is unclear. Finally, there is a fledgling system that proposes to use a combination of GPUs and FPGAs for cryptoanalytic problems [39], although to date the applications have only been tested on GPUs.

We also would like the system to be scalable if possible, although the focus of this paper is the combined heterogeneity of the hardware accelerators in a single-node architecture. The main problems we are concerned with here feature embarrassingly parallel analysis pipelines, and so extrapolation to a cluster system ought to be relatively straight forward. The granularity of this system is dictated by the particular algorithm being used; the most coarse-grained pipelines will be those with a large reliance on GPU-based resources.

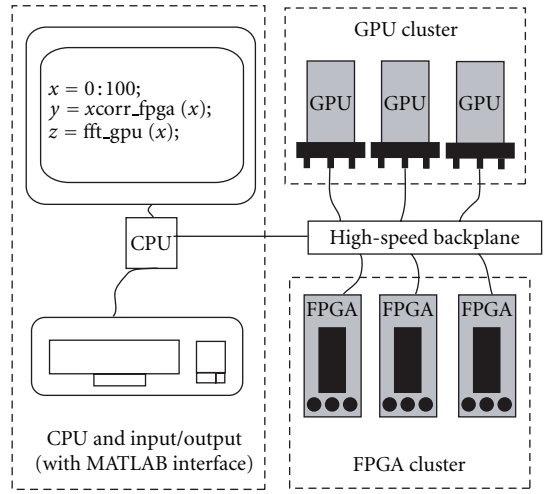



Figure 1: A schematic of our original concept for a heterogeneous CPU/GPGPU/FPGA system.

Table 1: Description of subsystem hardware configuration for the "Chimera" heterogeneous CPU/GPGPU/FPGA computing platform used here.

| Subsystem | Vendor | Model |
|---|---|---|
| CPU | Intel | i7 Hexacore |
| GPGPU | nVidia | Tesla C2070 |
| FPGA | Altera | Stratix-IV |

Finally, a significant constraint was an inexpensive initial outlay, which immediately restricted us to commercial-off-the-shelf (COTS) components. Table 1 lists the components of the heterogeneous system we describe here, which we call "Chimera," after the mythical Greek beast with a head of a goat, a snake, and a lion on the same body.

Aside from the actual CPU, FPGA, and GPU components, perhaps the most important element in this system is the high-speed backplane or interconnect. Because of COTS constraints, we eventually settled on the simplest solution, that is, that already residing as the northbridge system on the CPU motherboard. In this case, the interconnect protocol is Peripheral Component Interconnect express (PCIe) Gen 2.0, on an Intel DX58SO2 motherboard. Although the board has 2X PCIe 2.0 $\times$ 16 ports, only one 16 lane port is dedicated, the other is multiplexed with a third 8 lane slot. The maximum theoretical PCIe throughput, is therefore, 2 $\times$ 16 lane devices or 32 GB/s. These 32 PCIe lanes are routed to the I/O hub processor (the 82X58IOH, which we loosely term here a "northbridge") which implements Intel's Quick Peripheral Interconnect (QPI) protocol to the CPU. The QPI has a 25.6 GB/s point-to-point link to the CPU. In spite of the impressive performance of the motherboard we found, as expected, the PCIe bottleneck presented the most significant limitation to our computing model. We choose to ignore this limitation in what follows for several reasons.

(1) The limitation is algorithm dependant, in some cases (e.g., generation of pseudorandom numbers) large-data sets are developed and processed solely on-chip. In other cases, processing pipelines may be organized to avoid this bottleneck.

(2) FPGA devices, in particular, are provided with very high speed I/O connections allowing multiple FPGAs to process and reduce data-sets before passing them to the final, PCIe limited device.

(3) The purpose of the Chimera is to prove the concept of the hybrid computing model using low-cost COTS devices. Having established that a hybrid design is limited only by interconnect speed the way is open for faster and more expensive interconnect solutions.

The Altera Stratix-IV FPGAs reside on DE-530 development boards, with an 8-lane PCIe interface, while the Tesla C2070 has a 16-lane PCIe interface. The development environment for the FPGAs was Verilog and ModelSim, while that for the GPUs was C/C++ for CUDA and nVidia's SDK libraries. Because there is not yet a widely available communication protocol that allows FPGA-GPU communication without the mediation of a CPU, we are currently developing kernel modules for the PCIe bus. A primary goal of the Chimera system is to provide access to high-performance computing hardware for novice users. This inherently means providing an operating system (OS) with familiar signal processing tools (e.g., MATLAB). Running the OS is naturally a task exclusively handled by the CPU, but it presents some difficulties because the security layers of the OS will usually deny direct access to the FPGA and GPU hardware. In the case of the GPU, this problem is solved by the vendor (nVidia) provided drivers, but, for the FPGA, an alternative approach is necessary: custom driver development is necessary. We opted for a Linux-based OS because we feel it is much simpler to develop drivers than for proprietary OSs. Like all modern OSs, GNU/Linux implements a virtual memory environment that prevents "user" code from directly accessing memory and memory-mapped peripherals. In order to directly control system hardware, such as a PCIe device, it is necessary to write code that runs in "kernel" mode. Linux permits these "kernel modules" to be loaded and unloaded into the running kernel *via* the `insmod` and `rmmod` commands. This provides a straightforward means to share data between the FPGA and GPU devices on the PCIe bus, as well as the system memory. A schematic of this stack design is given in Figure 2.

It is also possible to rebuild the Linux kernel, thus incorporating the module into a custom Linux kernel. The kernel code then provides a bridge between the user code space and the FPGA hardware. As these kernel modules are still under development, and the bottleneck from the PCIe transfer is simple to calculate for the simple examples below, we consider here the subsystem performance only. Hence, the run-time and data-transfer systems are currently fairly primitive. In order to optimize performance, the parallel programming models are algorithm dependent, including the number of threads and how data is shared. Data transfer between the subsystem components currently has only limited automation, and pipelines are mediated by the CPU, which is highly undesirable. However, we are in the process of building data transfer protocols (*via* the kernel modules) that depend upon the particular application but are implemented using a coherent level of abstraction.

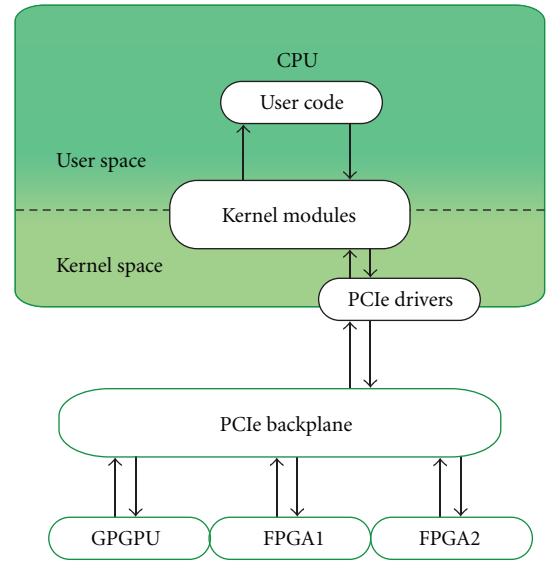



Figure 2: The software stack hierarchy of the Chimera system. The Linux kernel modules provide an interface between user-generated code and the PCIe bus and hardware components.

Finally, this system was conceived with the explicit consideration of the significant trade-off between development time (especially associated with development of the FPGA related pipeline and PCIe modules) and performance. Hence, we consider only computing solutions that are likely to have a high reuse within the data analysis community.

## 5. Appropriate Algorithms for a Heterogeneous Computing System

As with most compute intensive problems, the implementation is extremely dependent on the underlying hardware and the most significant bottlenecks. Much consideration is required in the "technology mapping" from one system to another. For example, an important consideration for FPGA HPC applications is whether fixed or floating point calculations are required. Data analysts have become accustomed to assume that their pipeline requires single precision or more, without considering that this may actually *decrease* the absolute error of the calculation. It is also not entirely true that FPGAs perform poorly on floating point performance [40] (note that this is not necessarily the same as conforming exactly to the IEEE-754 single or double precision definition [41]; GPUs do not natively support denormals but now perform linear operations very well with most double precision calculations). Considering around 50% of FPGA logic can be consumed by tracking denormals, there are recent ingenious developments, optimizing floating point pipelines on FPGAs, such as Altera's "fused datapath" approach [42].

We consider here three important algorithms to attempt to illustrate the considerations required, and relative benefits of, mapping common problems to our heterogeneous computing system.

*5.1. Monte Carlo Integration.* The simplest illustration of the advantages of this system is the well-known Monte Carlo calculation of $\pi$ (the ratio of the circumference of a circle to its diameter) [43]. This proceeds as follows. A large set of points $(x, y)$ are created from pairs of random numbers, uniformly distributed between $-1$ and $+1$, such that they fall within a square of area $2 \times 2 = 4$ units$^2$. Each point is then checked to see if it lies within a circle of unit radius (equivalently solving the inequality $x^2 + y^2 < 1$). The ratio of the number of points satisfying this inequality to the total number of points then determines the ratio of the total area to the area of the circle. Yet the area of a circle of unit radius is $\pi$, so the ratio will yield a numerical estimate of $\pi/4$.

Because FPGAs have the unrivalled capacity to generate large quantities of uniformly distributed random numbers, and GPUs are vastly superior at taking squares (or square roots) over any other commonly available general purpose platform, it would be sensible if an FPGA were to generate the randomly placed points (in parallel), which were in turn directly given to a GPU to determine the placement of each point (i.e., within the circle or not). This calculation and how we anticipate performing it using a hybrid approach is depicted in Figure 3. We could implement a more sophisticated quasirandom version of this algorithm, that is, using a Halton, Sobol' or other low-discrepancy sequence [26, 43], but considering this a rather poor way to calculate $\pi$, we restrict ourselves here to uniform pseudorandom distributions for illustrative purposes.

The pseudorandom calculation of $\pi$ was implemented entirely in the GPU and entirely in the FPGA. In both cases, two 32 bit unsigned integers ($(x, y)$ pairs) were generated using a pseudorandom generator. These were squared and added and the result compared to unity. The limiting factor in the StratixIV530 FGPA was the available number of multiplier blocks which were necessary for the square operation. Our design required 5 DSP18 blocks per sampling module, allowing a total of $\approx$200 parallel units. Applying a conservative clock speed of 120 MHz, we achieved 24 giga-samples per second. Surprisingly, this is approximately an order of magnitude greater than our results for the GPU, which performed 100,000 trials in 47 $\mu$sec, giving $\approx$2.13 giga-samples per second. These results agree broadly with those in [26], in that the FPGA calculation of the entire pipeline is about an order of magnitude faster than the GPU implementation, and many many times faster than that by a comparable CPU.

This result ought to be surprising, *prima facie*, considering that the multiplication intensive calculations involved in testing whether the points lie within the unit circle ought to favor the GPU. However, one must remember that a considerable amount of effort went into an implementation optimized for the FPGA, including the avoidance of the costly square root operation. We expect the GPU to perform a lot better, in relative terms, when the function to be integrated is more complicated than this extremely simple model. This argument, of course, ignores the fact that copying memory to devices takes far longer than these simple calculations (e.g., in the case of the GPU, about 42 times as long, at 2,118 $\mu$sec).

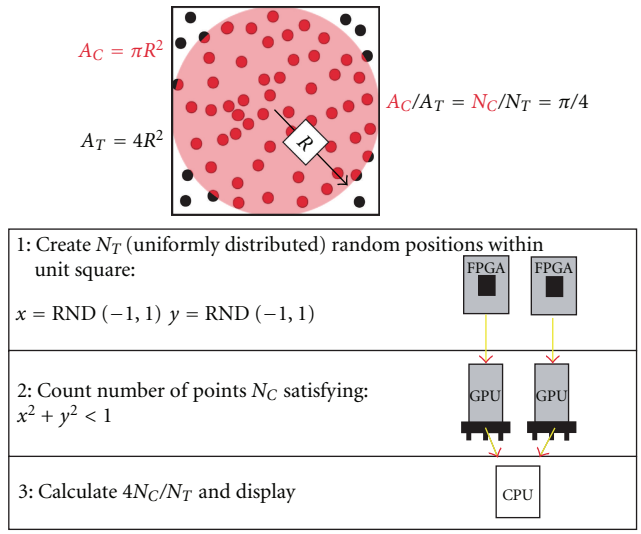



FIGURE 3: A schematic of Monte Carlo calculation of $\pi$, and how we expect to perform this calculation on a hybrid GPU/FPGA system.

*5.2. Normalized Cross-Correlation.* Template matching is one of the most important tasks in signal processing and is often achieved by computing the cross-correlation (or "sliding-dot product") between the template and a search signal. For example, cross-correlation is a crucial operation for resolving images in synthetic aperture array-based observatories such as the Very Long Baseline Array or the proposed Square Kilometre Array (SKA) [44].

The point in the search signal with maximum correlation is considered the best match to the template. For discrete one-dimensional signals, we seek to find $\max(A \times B[t])$, where $A \times B[t] = \sum_T A[T] * B[T + t]$. Essentially, we are treating the template, and a template-sized window of the search signal, as vectors in $N$-dimensional space (where $N$ is the length of the template) and computing their dot-product. The Pythagorean relationship shows that, for vectors of equal length, this is simply a measure of the angular relationship between them, since $\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}| \cdot |\mathbf{B}| \cos\theta$.

Normalized cross-correlation (NCC) is so called because it divides the cross-correlation by the product of the magnitude of the vectors, thus calculating $\cos\theta$, regardless of the length of the vectors. NCC is widely applied in image processing, such as video compression, pattern recognition, and motion estimation where the signals are, of course, two-dimensional, real, and nonnegative. For image processing applications, the 2DNCC is given by (1):

$$\sum_X \sum_Y \frac{A[x, y] \cdot B[x + X, y + Y]}{\sqrt{A^2 B[X, Y]^2}}. \quad (1)$$

Assuming the pixel data in question is always an integer and, since $\sqrt{A^2B^2} \leq A^2B^2$, we can equally express (1) as (2):

$$\sum_X \sum_Y \frac{A[x,y] \cdot B[x+X, y+Y]}{A^2 B[X,Y]^2}, \tag{2}$$

provided we are only interested in finding the peak of the NCC surface.

Computationally, there are many options to accelerate (2) [45]. For most real-world datasets, calculation of the numerator is faster if the Fourier shift theorem is applied and the (unnormalized) cross-correlation computed via the usual transform multiply and inverse transform approach (although it is important to remember that in order to prevent wrap-around pollution it is necessary to zero-pad both images to be size($A$) + size($B$) − 1). The multiply-intensive nature of this approach, combined with the all-but-essential use of floating-point data types, leads us to conclude that the numerator will be best computed with the GPU.

The ideal platform to calculate the denominator, however, is not so easily assured. As the image data is integral and the only operators are the sum and square, which are both closed under the set of integers, this would seem an ideal candidate for an FPGA. However, because $\mathbf{B}^2 \equiv \mathbf{B} \cdot \mathbf{B}$, it is also a task which might be well handled by a second GPU.

We evaluated the 2D NCC using a common image matching task. We exhaustively searched a $1024 \times 768$ pixel search image for an $8 \times 8$ template. In both cases, the pixels used were 16 bit unsigned greyscale integers. The numerator of (2) was calculated in the Fourier domain on the C2070 GPU and found to take 6.343 ms (≈158 frame/s). We then investigated two approaches to calculating the denominator: FPGA based, or using a second GPU. The FPGA implementation of the dot-product relied on the multiply-accumulate pipeline built into the DSP18 blocks in the StratixIV FPGA. Four DSP18s were required for each $8 \times 8$ dot-product block and, because very few of the FPGA's other resources were required, this was the limiting factor in determining the number of parallel units. For the StratixIV530 device, this allowed 256 dot-product modules to operate in parallel. As the DSP18 is a hard silicon block, the maximum clock speed was relatively high, at slightly over 400 MHz, giving a total value of roughly 10 giga-ops. For the search image size quoted, this would achieve ≈12.5 kframes/s. The GPU dot-product was profiled at one operation in about 1.4 nsec, or 715 Mop/s, corresponding to a frame rate of ≈894 frames/s.

Thus, in this implementation, ignoring the impact of the PCIe bottleneck, a hybrid system comprising a GPU working in tandem with an FPGA was found to achieve a better result than a system consisting of two GPUs. We would like to apply a similarly optimized algorithm to the correlation problem required by a synthetic aperture array observatory such as the VLBA or SKA [44].

*5.3. Continuous Gravitational Wave Data Analysis.* A much more complicated, and consequently useful, example is that related to the computationally bound problems found in gravitational wave data analysis. A number of mechanisms may cause rotating neutron stars to emit periodic distortions of space time (gravitational waves), which may be detected by ground-based gravitational wave observatories such as LIGO [46] or Virgo [47]. The data analysis pipeline is as follows: the data coming from the gravitational wave observatories is filtered, then template matching is applied. If no candidate is found, in order to determine the statistical upper limits, intensive Monte Carlo calculation of injections is required. This stage is so computationally intensive that it is often not fully implemented because of the prohibitive computational cost [48]. We estimate the Chimera platform will be able to provide these limits for approximately a dozen potential neutron star targets, including central objects in supernova remnants of unknown pulsation frequency. There are already GPU-based acceleration of similar pipelines used in gravitational wave data analysis; many FFTW3-based routines have been replaced with CUDA FFT, to enable low-latency detection of gravitational waves from coalescing neutron stars [19]. We see a similar pipeline as a natural application for the Chimera system we describe here, and we are in the process of implementing this.

*5.4. Other Promising Algorithms.* Many promising data analysis applications that are considered computationally expensive may be implemented rather simply using this type of heterogeneous hardware acceleration. For example, digital filter application is efficiently implemented on FPGA devices [49]. A number of promising analysis techniques based on compressed sensing [50] are considered computationally expensive. However, the most compute-intensive component, namely, the least-squares minimization routine, may be efficiently implemented on an FPGA via Cholesky decomposition [51–53].

## 6. Potential for Other Data Analysis Applications: Analysis via Berkeley's "Thirteen Dwarves"

Probably the most labor-intensive process involved in choosing the most appropriate platform weighting between the hardware accelerators in a heterogeneous system is that of identifying the most appropriate algorithms—especially their most efficient implementations—for a given pipeline and input/output constraints. Although we have identified a small number of algorithms here, it would be an interesting and valuable exercise to consider the possible classifications to determine which hardware acceleration combinations would be most appropriate.

In practice, there is a natural classification of problems merely by virtue of the similarity in computation and data movement through usage of the same software packages. Take, for example, FFTW/FFTW3 for spectral methods, or the dense linear algebra LAPACK/ScaLAPACK libraries. Indeed, the latter forms an important benchmark providing some measure of CPU performance (and possible entry into the popular "Top 500" list). However, we would like a more systematic approach to benchmark performance on parallel algorithms that are not necessarily strongly dependent on linear algebra.

In order to identify the likely future requirements of software and hardware design, Phil Colella identified seven parallel numerical methods important for science and engineering [54], known as "dwarves" (a reference to the Snow White fairy tale). This concept of a dwarf, as an "algorithmic method encapsulating a pattern of computation and/or communication," was further extended by the Department of Computer Science of UC Berkeley, to thirteen [55]. It is intended these "dwarves" comprise an algorithm classification system spanning the entire landscape of parallel computing for the foreseeable future ([55]; see Table 2). We see this approach as a promising means of determining the relative (dis)advantages of our Chimera system on other scientific problems. To our knowledge, this is the first attempt at the analysis of dwarf performance on systems heterogeneous at the hardware accelerator level.

The example of the Monte Carlo calculation of $\pi$ above falls under the "MapReduce" Dwarf: the Mapper function is each trial point, while the reducer just aggregates the counts. From the Chimera perspective, the mapper functions are either performed on the FPGA (the simple example above) or split between the FPGAs and the GPU, while the reducer runs on the CPU. The normalized cross-correlation would be classified within the "Dense Matrix" dwarf. The much more complicated case of the full gravitational wave analysis pipeline largely falls under the "Spectral Methods" jurisdiction, along with that of "MapReduce."

It is clear that many dwarves naturally map to each of the separate hardware accelerators. For example, the "Dense Matrix" dwarf equivalently relates to LAPACK/ScaLAPACK performance on a problem such as principal component analysis of a dense structure. Here, we should expect different performance for floating and fixed point operations, and hence we expect the GPU to excel alone on floating point (at least for matrices of up to rank 4 [29]), while the FPGA will be extremely competitive for fixed point versions. The same argument applies for naïve implementations of the "Sparse Matrix" dwarf, we expect GPUs to have superior performance calculating a sparse PCA problem in floating point, while the FPGA ought to well on fixed point. The "Spectral" dwarf generally comprises an FFT-based computation, such as wavelet decomposition. It is difficult for any platform to beat the GPU CUFFT library, and hence the GPU will be superior in most implementations, although for large numbers of FFT points, FPGAs may be more appropriate [40].

On the other hand, "Combinational Logic" problems (dwarf 8) such as hashing, DES encryption, or simulated annealing, are extremely well suited for the logic-intensive FPGA. It is also clear the "Finite State Machine" dwarf (13), such as control systems, compression (e.g., the bzip2 function), or cellular automata, can be most easily optimized by an FPGA. For example, consider a simple implementation of a 4-bit TTL (Transistor-Transistor Logic) counter, requiring 4 XOR gates, 3 AND gates, and 4 1-bit registers, our Stratix-4 could produce $\approx$120 k operations per clock cycle, or roughly 36 peta-op/s.

What are not so clear are problems requiring conditional elements or communication between hardware accelerators that would require prohibitively costly transfers across the back plane. The "N-Body" dwarf generally consists of calculations such as particle-particle interactions. A Barnes-Hut-based particle-particle N-Body model, as used for modelling astrophysical gravitating systems [56], would be able to calculate the changes in interaction on a GPU, while the memory-intensive cell (spatial position) data would be optimally handled by an FPGA. Although there is an implementation of an Ising "spin-flip" model on the Janus FPGA cluster [25], an example of a "Structured Grid" dwarf, a simple Ising model with a limited number of FPGAs would be better optimized using a GPU in addition.

The "Unstructured Grid" dwarf involves Adaptive Mesh Refinement, where calculations may be simplified by considering that only salient points in a space need be calculated, such as for adaptive finite element modelling (FEM) or computational fluid dynamics (CFD). The CPU will be useful in this case to tally changes in the mesh and co-ordinate calculations on salient mesh points using the FPGA and/or GPU.

Because of its conditional nature, "graph traversal" (including selection (Section 3.4, [43]), searching (Section 8.5, [43]) or decision trees) requires coordination from a CPU, and hardware acceleration is dependent on the particular application. "Dynamic Programming," such as the famous "Tower of Hanoi" problem, also requires CPU coordination, and also memory-intensive routines that are likely to benefit from an FPGA.

"Backtrack/Branch-and-Bound" problems, including search and global optimization problems, also depend on coordination from a CPU and again are generally memory intensive. "Graphical Models," such as Hidden Markov, probabilistic, neural, and Bayesian networks are also heavily dependent on coordination.

In light of these arguments, we summarize the most promising subsystem combination to apply for each dwarf

Table 2: The "Thirteen Dwarves" of Berkeley. Each dwarf represents an "algorithmic method encapsulating a pattern of computation and/or communication," and this intended to be a comprehensive list of the major requirements of parallel computational problems for now into the short term.

| | Dwarf | Examples/Applications |
|---|---|---|
| 1 | Dense Matrix | Linear algebra (dense matrices) |
| 2 | Sparse Matrix | Linear algebra (sparse matrices) |
| 3 | Spectral | FFT-based methods |
| 4 | N-Body | Particle-particle interactions |
| 5 | Structured Grid | Fluid dynamics, meteorology |
| 6 | Unstructured Grid | Adaptive mesh FEM |
| 7 | MapReduce | Monte Carlo integration |
| 8 | Combinational Logic | Logic gates (e.g., Toffoli gates) |
| 9 | Graph traversal | Searching, selection |
| 10 | Dynamic Programming | Tower of Hanoi problem |
| 11 | Backtrack/ Branch-and-Bound | Global optimization |
| 12 | Graphical Models | Probabilistic networks |
| 13 | Finite State Machine | TTL counter |

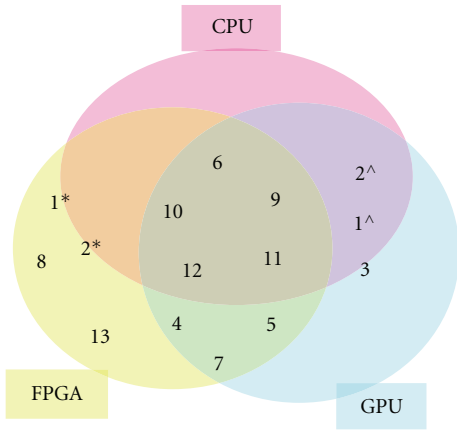



Figure 4: The most appropriate hardware acceleration subsystem combination for representative problems from the "Thirteen Dwarves" (Table 2). The * refers to fixed point, while ^ represents floating point calculations.

as a Venn diagram in Figure 4. Of course, this should be understood that the performance is heavily dependent on the particular implementation, generation of subsystem (including on-board memory, number of LUTs, etc.), and interconnect speed.

To characterize the performance of the Chimera system, we would like to analyze representative problems in each dwarf in the future using appropriate benchmark packages such as, for example, the "DwarfBench" package [57] or "Rodinia" [58]. Such a benchmark would be of use to researchers considering the benefits of this approach for their own work.

## 7. Discussion

We see a heterogeneous CPU/GPU/FPGA solution as a viable future platform for ameliorating some of the computationally bound problems in astronomy based on performance, initial outlay, and power consumption considerations. Because of the outstanding progress made in microprocessor technologies, most members of the astronomical and broader scientific community will not have considered the possibility of including an FPGA-based accelerator to their analysis pipelines; indeed the adoption of the GPU is relatively new within the data analysis community.

However, this solution is not without challenges. There is a significant investment required in development time. This begins with the "technology mapping" stage, which can require many hours for a modest pipeline, such as in Sections 5.1 and 5.2, each of which required considerable thought of how to distribute compute resources, including small-scale trials using MATLAB as a development environment, comprising one to two hours each. The GPU programming for these two examples took a total of about five hours, largely thanks to nVidia's comprehensive CUDA SDK support. However, because the degree of optimization of the FPGA pipelines meant the total amount of time spent "programming" was approximately forty to fifty hours, the majority of which was spent in the synthesis/debugging phase using ModelSim. Although we have not yet implemented the example in Section 5.3—which requires the dynamic range given by floating point types—considerable development time was saved using Altera's DSP Builder, which interfaces easily with Mathworks' Simulink. Another issue to be aware of is that a high degree of hardware optimization generally means a trade-off in problem-size scalability. For example, in Section 5.1 example, the pairs of numbers $(x, y)$ are each 32-bit and the cases under-/overflow are well known; a naïve scaling to, say, single precision floating point pairs would quickly reduce the number of parallel units on the FPGA. In example Section 5.2, if instead of an $8 \times 8$ template, we were to use $32 \times 32$ template, performance would scale very poorly because the limiting factor in performance is the number of DSP18 blocks on the StratixIV530 FPGA.

A promising advantage of this platform is the low power consumption. Taking the peak rated single precision performance of each subsystem gives 1.02 Tflop/s for the Tesla C2070, drawing a power of 228 W. This gives 4.3 Gflop/s/W, or 4.3 Gflop/J. Compare to the StratixIV, with 500 Mflop performance, drawing around 20 W at this performance, yielding roughly 25 Mflop/s/W. Perhaps more importantly, the FPGA draws virtually no power when not performing intensive calculations, unlike both the GPU and CPU systems.

Although we have shown the merits and challenges of a mixed system, the advantages are obvious for a diverse range of parallel computing tasks, as shown by analysis of Berkeley's "Thirteen Dwarves." This paper hopefully provides a blueprint for future researchers intending to perform computationally intensive investigations and are willing to embrace a heterogeneous computing platform.

## Acknowledgments

The authors wish to thank the Australian National University and Altera Corporation for their generous support. R. Inta is supported by an Australian Research Council "Discovery" project and D. J. Bowman is supported by an Australian Research Council "Super Science" Project. They would like also to thank Martin Langhammer for helpful discussion, and the anonymous reviewers, whose suggestions greatly improved the manuscript.

## References

[1] B. P. Schmidt, N. B. Suntzeff, M. M. Phillips et al., "The high-Z supernova search: measuring cosmic deceleration and global curvature of the universe using type Ia supernovae," *Astrophysical Journal*, vol. 507, no. 1, pp. 46–63, 1998.

[2] G. F. Smoot, C. L. Bennett, A. Kogut et al., "Structure in the COBE differential microwave radiometer first-year maps," *Astrophysical Journal*, vol. 396, no. 1, pp. L1–L5, 1992.

[3] D. N. Spergel, L. Verde, H. V. Peiris et al., "First-year Wilkinson Microwave Anisotropy Probe (WMAP) observations: determination of cosmological parameters," *Astrophysical Journal, Supplement Series*, vol. 148, no. 1, pp. 175–194, 2003.

[4] University of California, 2011, http://setiathome.berkeley.edu/.

[5] B. Knispel, B. Allen, J. M. Cordes et al., "Pulsar discovery by global volunteer computing," *Science*, vol. 329, no. 5997, p. 1305, 2010.

[6] T. Cornwell and B. Humphreys, "Data processing for ASKAP and SKA," 2010, http://www.atnf.csiro.au/people/tim.cornwell/presentations/nzpathwaysfeb2010.pdf.

[7] M. Hilbert and P. López, "The world's technological capacity to store, communicate, and compute information," *Science*, vol. 332, no. 6025, pp. 60–65, 2011.

[8] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top 500 supercomputers," 2011, http://www.top500.org/.

[9] LSST Corporation, "Large synoptic survey telescope," 2011, http://www.lsst.org/lsst/.

[10] The Australian National University, "The SkyMapper survey telescope," 2010, http://msowww.anu.edu.au/skymapper/.

[11] Pico Computing, Inc., "Using FPGA clusters for fast password recovery," 2010, http://www.scribd.com/doc/26191199/Using-FPGA-Clusters-for-Fast-Password-Recovery.

[12] C. Duhigg, "Stock traders find speed pays, in milliseconds," 2009, http://www.nytimes.com/2009/07/24/business/24trading.html.

[13] J. Ericson, "Processing Avatar," 2009, http://www.information-management.com/newsletters/avatar_data_processing-10016774-1.html.

[14] J. Jackson, "Supercomputing top500 brews discontent," 2010, http://www.pcworld.idg.com.au/article/368598/supercomputing_top500_brews_discontent/.

[15] N. Hasasneh, I. Bell, C. Jesshope, W. Grass, B. Sick, and K. Waldschmidt, "Scalable and partitionable asynchronous arbiter for micro-threaded chip multiprocessors," in *Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS '06)*, W. Grass, B. Sick, and K. Waldschmidt, Eds., vol. 3894 of *Lecture Notes in Computer Science*, pp. 252–267, Frankfurt, Germany, 2006.

[16] P. E. Ross, "A computer for the clouds," 2008, http://spectrum.ieee.org/computing/hardware/a-computer-for-the-clouds.

[17] M. Wehner, L. Oliker, and J. Shalf, "Low-power supercomputers (ieee spectrum)," 2009, http://spectrum.ieee.org/computing/embedded-systems/lowpower-supercomputers.

[18] K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *Proceedings of the 12th International Symposium on Field-Programmable Gate Arrays (FPGA '04)*, pp. 171–180, New York, NY, USA, February 2004.

[19] S. K. Chung, L. Wen, D. Blair, K. Cannon, and A. Datta, "Application of graphics processing units to search pipelines for gravitational waves from coalescing binaries of compact objects," *Classical and Quantum Gravity*, vol. 27, no. 13, Article ID 135009, 2010.

[20] B. R. Barsdell, D. G. Barnes, and C. J. Fluke, "Analysing astronomy algorithms for graphics processing units and beyond," *Monthly Notices of the Royal Astronomical Society*, vol. 408, no. 3, pp. 1936–1944, 2010.

[21] M. Bergano, F. Fernandes, L. Cupido et al., "Digital complex correlator for a C-band polarimetry survey," *Experimental Astronomy*, vol. 30, no. 1, pp. 23–37, 2011.

[22] L. De Souza, J. D. Bunton, D. Campbell-Wilson, R. J. Cappallo, and B. Kincaid, "A radio astronomy correlator optimized for the Xilinx Virtex-4 SX FPGA," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 62–67, August 2007.

[23] B. Klein, S. D. Philipp, I. Krämer, C. Kasemann, R. Güsten, and K. M. Menten, "The APEX digital fast fourier transform spectrometer," *Astronomy and Astrophysics*, vol. 454, no. 2, pp. L29–L32, 2006.

[24] J. Frigo, D. Palmer, M. Gokhale, and M. Popkin Paine, "Gamma-ray pulsar detection using reconfigurable computing hardware," in *Proceedings of the 11th IEEE Symposium Field-Programmable Custom Computing Machines*, pp. 155–161, Washington, DC, USA, 2003.

[25] F. Belletti, M. Guidetti, A. Maiorano et al., "Janus: an FPGA-based system for high-performance scientific computing," *Computing in Science and Engineering*, vol. 11, no. 1, Article ID 4720223, pp. 48–58, 2009.

[26] T. Xiang and B. Khaled, "High-performance quasi-Monte Carlo financial simulation: FPGA vs. GPP vs. GPU," in *Proceedings of the ACM Transactions on Reconfigurable Technology and Systems*, vol. 3, pp. 26:1–26:22, New York, NY, USA, 2010.

[27] M. Awad, "FPGA supercomputing platforms: a survey," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 564–568, Prague, Czech Republic, 2009.

[28] D. B. Thomas, L. Howes, and W. Luk, "A comparison of CPUs, GPUs, FPGAs, and massively processor arrays for random number generation," in *Proceedings of the 7th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '09)*, pp. 63–72, Monterey, Calif, USA, 2009.

[29] B. Cope, P. Y. K. Cheung, W. Luk, and S. Witt, "Have GPUs made FPGAs redundant in the field of video processing?" in *Proceeding of the IEEE International Conference on Field Programmable Technology*, vol. 1, pp. 111–118, December 2005.

[30] B. Cope, "Implementation of 2D convolution on FPGA, GPU and CPU," Tech. Rep., Imperial College, London, UK, 2006.

[31] D. H. Jones, A. Powell, C. -S. Bouganis, and P. Y.K. Cheung, "GPU versus FPGA for high productivity computing," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 119–124, 2010.

[32] D. Yang, J. Sun, J. Lee et al., "Performance comparison of cholesky decomposition on GPUs and FPGAs," in *Proceedings of the Symposium Application Accelerators in High Performance Computing (SAAHPC '10)*, Knoxville, Tenn, USA, 2010.

[33] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with GPUs and FPGAs," in *Proceedings of the Symposium on Application Specific Processors (SASP '08)*, pp. 101–107, Anaheim, Calif, USA, 2008.

[34] S. J. Park, D. R. Shires, and B. J. Henz, "Coprocessor computing with FPGA and GPU," in *Proceedings of the Department of Defense High Performance Computing Modernization Program: Users Group Conference—Solving the Hard Problems*, pp. 366–370, Seattle, Wash, USA, 2008.

[35] R. Inta and D. J. Bowman, "An FPGA/GPU/CPU hybrid platform for solving hard computational problems," in *Proceedings of the eResearch Australasia*, Gold Coast, Australia, 2010.

[36] M. Showerman, J. Enos, A. Pant et al., "QP: a heterogeneous multi-accelerator cluster," in *Proceedings of the 10th LCI International Conference on High-Performance Cluster Computing*, vol. 7800, pp. 1–8, Boulder, Colo, USA, 2009.

[37] K. H. Tsoi and W. Luk, "Axel: a heterogeneous cluster with FPGAs and GPUs," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA '01)*, pp. 115–124, Monterey, Calif, USA, 2010.

[38] E. J. Kelmelis, J. P. Durbano, J. R. Humphrey, F. E. Ortiz, and P. F. Curt, "Modeling and simulation of nanoscale devices with a desktop supercomputer," in *Proceedings of the Nanomodeling II*, vol. 6328, p. 62270N, 2006.

[39] W. Kastl and T. Loimayr, "A parallel computing system with specialized coprocessors for cryptanalytic algorithms," in *Proceedings of the Sicherheit*, pp. 73–83, Berlin, Germany, 2010.

[40] K. D. Underwood and K. S. Hemmert, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing*, chapter 31, Morgan Kaufmann Publishers, Burlington, Mass, USA, 2008.

[41] K. Asanovic, "IEEE standard for binary floating-Point Arithmetic," Tech. Rep. ANSI/IEEE Std., IEEE Standards Board, The Institute of Electrical and Electronics, 1985.

[42] M. Langhammer, "Floating point datapath synthesis for FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 355–360, Heidelberg, Germany, 2008.

[43] S. A. T. W. H. Press, W. T. Vettering, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, New York, NY, USA, 2nd edition, 1997.

[44] P. J. Napier, D. S. Bagri, B. G. Clark et al., "Very long baseline array," *Proceedings of the IEEE*, vol. 82, no. 5, pp. 658–672, 1994.

[45] D. Bowman, M. Tahtali, and A. Lambert, "Rethinking image registration on customizable hardware," in *Image Reconstruction from Incomplete Data VI*, vol. 7800 of *Proceedings of SPIE*, San Diego, Calif, USA, 2010.

[46] LIGO Scientific Collaboration, 2010, http://www.ligo.caltech.edu/.

[47] Virgo Scientific Collaboration, 2010, http://www.virgo.infn.it/.

[48] P. K. Patel, *Search for gravitational waves from a nearby neutron star using barycentric resampling*, Ph.D. thesis, California Institute of Technology, Pasadena, Calif, USA, 2011.

[49] D. Llamocca, M. Pattichis, and G. A. Vera, "Partial reconfigurable FIR filtering system using distributed arithmetic," *International Journal of Reconfigurable Computing*, vol. 2010, Article ID 357978, 14 pages, 2010.

[50] E. J. Candès, J. K. Romberg, and T. Tao, "Stable signal recovery from incomplete and inaccurate measurements," *Communications on Pure and Applied Mathematics*, vol. 59, no. 8, pp. 1207–1223, 2006.

[51] O. Maslennikow, V. Lepekha, A. Sergiyenko, A. Tomas, and R. Wyrzykowski, *Parallel Implementation of Cholesky $LL^T$—Algorithm in FPGA-Based Processor*, Springer, Berlin, Germany, 2008.

[52] D. Yang, H. Li, G. D. Peterson, and A. Fathy, "Compressed sensing based UWB receiver: hardware compressing and FPGA reconstruction," in *Proceedings of the 43rd Annual Conference on Information Sciences and Systems (CISS '09)*, pp. 198–201, Baltimore, Md, USA, 2009.

[53] A. Septimus and R. Steinberg, "Compressive sampling hardware reconstruction," in *Proceedings of the IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems (ISCAS '10)*, pp. 3116–3119, Paris, France, 2010.

[54] P. Colella, *Defining Software Requirements for Scientific Computing*, DARPA HPCS, 2004.

[55] K. Asanovic and U C Berkeley Computer Science Deptartment, "The landscape of parallel computing research: a view from Berkeley," Tech. Rep. UCB/EECS-2006-183, UC Berkeley, 2005.

[56] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, no. 6096, pp. 446–449, 1986.

[57] R. Palmer et al., "Parallel dwarfs," 2011, http://paralleldwarfs.codeplex.com/.

[58] S. Che, M. Boyer, J. Meng et al., "Rodinia: a benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '09)*, pp. 44–54, Austin, Tex, USA, October 2009.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 507173, 16 pages
doi:10.1155/2012/507173

*Research Article*

# Throughput Analysis for a High-Performance FPGA-Accelerated Real-Time Search Application

**Wim Vanderbauwhede,[1] S. R. Chalamalasetti,[2] and M. Margala[2]**

[1] *School of Computing Science, University of Glasgow, Glasgow, G12 8QQ, UK*
[2] *Department of Electrical and Computer Engineering, University of Massachusetts Lowell, Lowell, MA 01854, USA*

Correspondence should be addressed to Wim Vanderbauwhede, wim@dcs.gla.ac.uk

Received 13 October 2011; Accepted 20 December 2011

Academic Editor: Miaoqing Huang



We propose an FPGA design for the relevancy computation part of a high-throughput real-time search application. The application matches terms in a stream of documents against a static profile, held in off-chip memory. We present a mathematical analysis of the throughput of the application and apply it to the problem of scaling the Bloom filter used to discard nonmatches.

## 1. Introduction

The focus on real-time search is growing with the increasing adoption and spread and of social networking applications. Real-time search is equally important in other areas such as analysing emails for spam or search web traffic for particular patterns.

FPGAs have great potential for speeding up many types of applications and algorithms. By performing a task in a fraction of the time of a conventional processor, large energy savings can be achieved. Therefore, there is a growing interest in the use of FPGA platforms for data centres. Because of the dramatic reduction in the required energy per query, data centres with FPGA search solutions could operate at a fraction of the power of current data centres, eliminating the need for cooling infrastructure altogether. As the cost of cooling is actually the dominant cost in today's data centres [1], the savings would be considerable. In [2, 3] we presented our initial work on applying FPGAs for acceleration or search algorithms. In this paper, we present a novel design for the scoring part of an FPGA-based high-throughput real-time search application. We present a mathematical analysis of the throughput of the system. This novel analysis is applicable to a much wider class of applications than the one discussed in the paper; any algorithm that performs nondeterministic concurrent accesses to a shared resource can be analysed using the model we present. In particular, the technology presented in this paper can also be used for "traditional," that is, inverted index based, web search.

## 2. Design of the Real-Time Search Application

Real-time search, in information retrieval parlance called "document filtering," consists of matching a stream of documents against a fixed set of terms, called the "profile." Typically, the profile is large and must therefore be stored in external memory.

The algorithm implemented on the FPGA can be expressed as follows.

(i) A *document* is modelled as a "bag of words," that is, a set $D$ of pairs $(t, f)$, where $f \triangleq n(t, d)$ is the *term frequency*, that is, number of occurrences of the term $t$ in the document $d$; $t \in \mathbb{N}$ is the *term identifier*.

(ii) The profile $M$ is a set of pairs $p = (t, w)$ where the *term weight* $w$ is determined using the "Relevance Based Language Model" proposed by Lavrenko and Croft [4].

In this work we are concerned with the computation of the document score, which indicates how well a document matches the profile. The document has been converted to the bag-of-words representation in a separate stage. We perform this stage on the host processor using the Open Source

information retrieval toolkit Lemur [5]. We note that this stage could also be very effectively performed on FPGAs.

Simplifying slightly, to determine if a document matches a given profile, we compute the sum of the products of term frequency and term weight

$$\sum_{i \in D} f_i w_i. \tag{1}$$

The weight is typically a high-precision word (64 bits) stored in a lookup table in the external memory. If the score is above a given threshold, we return the document identifier and the score by writing it into the external memory.

*2.1. Target Platform.* The target platform for this work is the Novo-G FPGA supercomputer [6] hosted by the NSF Center for high-performance reconfigurable computing (CHREC) (http://www.chrec.org/). This machine consists of 24 compute servers which each host a GiDEL PROCStar-III board. The board contains 4 FPGAs with 2 banks of DDR SDRAM per FPGA used for the document collection and one for the profile. The data width is 64 bits, which means that the FPGA can read 128 bits per memory per clock cycle [7]. For more details on the platform, see Section 4.

*2.2. Term-Scoring Algorithm.* To simplify the discussion, we first consider the case where terms are scored sequentially, and that, as in our original work, we use a Bloom filter to limit the number of external memory accesses.

For every term in the document, the application needs to look up the corresponding profile term to obtain the term weight. As the profile is stored in the external SDRAM, this is an expensive operation (typically 20 cycles per access). The purpose of document filtering is to identify a small amount of relevant documents from a very large document set. As most documents are not relevant, most of the lookups will fail (i.e., most terms in most documents will not occur in the profile). Therefore, it is important to discard the negatives first. For that purpose we use a "trivial" Bloom filter implemented using the FPGA's on-chip memory.

*2.2.1. "Trivial" Bloom Filter.* A Bloom filter [8] is a datastructure used to determine membership of a set. False positives are possible, but false negatives are not. With this definition, the design we use to reject negatives is a Bloom filter. However, in most cases, a Bloom filter uses a number ($k$) of hash functions to compute several keys for each element in the set and adds the element to the table (assigns a "1") if element is in the set. As a result, hash collisions can lead to false positives.

Our Bloom filter is a "trivial" edge case of this more general implementation; our hashing function is the identity function $key = elt$, and we only use a single hash function ($k = 1$) so every element in the set corresponds to exactly one entry in the Bloom filter table. As a result, the size of the Bloom filter is the same as the size of the set, and there are no false positives. Furthermore, no elements are added to the set at run time.

*2.2.2. Bloom Filter Dimensioning.* The internal block RAMs of the Altera Stratix-III FPGA that support efficient single-bit access are limited to 4 Mb; on a Stratix-III SE260, there are 864 M9K blocks that can be configured as $8\,\mathrm{K} \times 1$ [9]. On the other hand, the vocabulary size of our document collection is 16 M terms (based on English documents using unigrams, diagrams, and trigrams). We therefore used a very simple "hashing function," $key = elt \gg 2$. Thus we obtain one entry for every four elements, which leads to three false positives out of four on average. This obviously results in a four times higher access rate to the external memory than if the Bloom filter would be 16 Mb. As the number of positives in our application is very low, the effect on performance is limited.

*2.2.3. Document Stream Format.* The document stream is a list of *(document identifier, document term pair set)* pairs. Physically, the FPGA accepts a fixed number $n$ of streams of words with fixed width $w$. The document stream must be encoded onto these word streams. As both elements in the document term pair $d_i = (t_i, f_i)$ are unsigned integers, $m$ pairs can be encoded onto a word if $w$ is larger than or equal to $m$ times the sum of the magnitudes of the maximum values for $t$ and $f$

$$w \geq m\left(\left\lceil \log_2 t_{\max} \right\rceil + \left\lceil \log_2 f_{\max} \right\rceil\right). \tag{2}$$

To mark the start a document we insert a header word (identified by $f = 0$) followed by the document ID.

*2.2.4. Profile Lookup Table Implementation.* In the current implementation, the lookup table that stores the profile is implemented in the most straightforward way; as the vocabulary size is $2^{24}$ and the weight for each term in the profile can be stored in 64 bits, a profile consisting of the entire vocabulary could be stored in the 256 MB SDRAM, which is less than the size of the fixed SDRAM on the PROCStar-III board. Consequently, there is no need for hashing, the memory contains zero weights for all terms not present in the profile.

*2.2.5. Sequential Implementation.* The diagram for the sequential implementation of the design is shown in Figure 1.

Using the lookup table architecture and document stream format as described above, the actual lookup and scoring system is quite straightforward, the input stream is scanned for header and footer words. The header word action is to set the document score to 0; the footer word action is to collect and output the document score. For every term in the document, first the Bloom filter is used to discard negatives, and then the profile term weight is read from the SDRAM. The score is computed and accumulated for all terms in the document, and finally the score stream is filtered against a threshold before being output to the host memory. The threshold is chosen so that only a few tens or hundreds of documents in a million are returned.

If we would simply look up every term in the external memory, the maximum achievable throughput would be

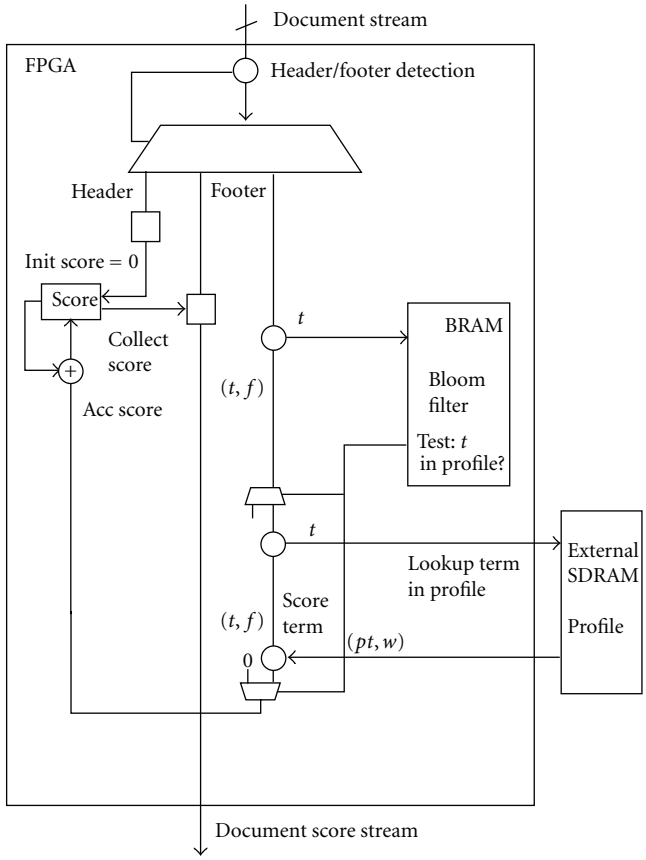



Figure 1: Sequential document term scoring.

$1/\Delta t_S$, with $\Delta t_S$ the number of cycles required to look up the term weight in the external memory and compute the term score. The use of a Bloom filter greatly improves the throughput as the Bloom filter access will typically be much faster than the external memory access and subsequent score computation. If the probability for a term to occur in the profile is $P_P$ and the access time to the Bloom filter is $\Delta t_B$, the average access time will become $\Delta t_B + P_P \cdot \Delta t_S$. In practice $P_P$ will be very low as most document terms will not occur in the profile (because otherwise the profile would match all documents). The more selective the profile, the fewer the number of document terms that match it.

*2.3. Parallelising Lookups.* The scoring process as described above is sequential. However, as in the bag-of-words representation all terms are independent, there is scope for parallelisation. In principle, all terms of a document could be scored in parallel, as they are independent and ordering is of no importance.

*2.3.1. Parallel Document Streams.* In practice, even without the bottleneck of the external memory access, the amount of parallelism is limited by the I/O width of the FPGA, in our case 64 bits per memory bank. A document term can be encoded in 32 bits (a 24-bit term identifier and an 8-bit term frequency). As it takes at least one clock cycle of the FPGA clock to read in two new 64-bit words (one per bank), the best case for throughput would be if 4 terms per document would be scored in parallel in a single cycle. However, in practice scoring requires more than one cycle; to account for this, the process can be further parallelised by demultiplexing the document stream into a number of parallel streams. If, for example, scoring would take 4 cycles, then by scoring 4 parallel document streams the application could reach the maximal throughput.

*2.4. Parallel Bloom Filter Design.* Obviously, the above solution would be of no use if there would be only a single, single-access Bloom filter. The key to parallelisation of the lookup is that because the Bloom filter is stored in on-chip memory, accesses to it can be parallelised by partitioning the Bloom filter into a large number of small banks. The combined concepts of using parallel streams and a partitioned Bloom filter are illustrated in Figure 2. To keep the diagram uncluttered, only the paths of the terms (Bloom filter addresses) have been shown.

Every stream is multiplexed to all $m$ Bloom filter banks; every bank is accessed through an $n$-port arbiter. It is intuitively clear that, for large numbers of banks, the probability of contention approaches zero, and hence the throughput will approach the I/O limit—or would if none of the lookups would result in an external memory access and score computation.

## 3. Throughput Analysis

In this section, we present the mathematical throughput analysis of the Bloom filter-based document scoring system. The analysis consists of four parts.

(i) In Section 3.1 we derive an expression to enumerate all possible access patterns for $n$ concurrent accesses to a Bloom filter built of $m$ banks and use it to compute the probability for each pattern.

(ii) In Section 3.2 we compute the average access time for each pattern, given that $n_H$ accesses out of $n$ will result in a lookup in the external memory. We consider in particular the cases of $n_H = 0$ and $n_H = 1$ and propose an approximation for higher values of $n_H$.

(iii) In Section 3.3 we compute the probability that $n_H$ accesses out of $n$ will result in a lookup in the external memory.

(iv) In Section 3.4, combining the results from Section 3.2 and Section 3.3, we compute the average access time over all $n_H$ for a given access pattern; finally, we combine this with the results from 3.1 to compute the average access time over all access patterns.

*3.1. Bloom Filter Access Patterns.* We need to calculate the probability of contention between $c$ accesses out of $n$, for a Bloom filter with $m$ banks. Each bank has an arbiter which sequentialises the contending accesses, so $c$ contending accesses to a given bank will take a time $c \cdot \Delta t_B$, with $\Delta t_B$

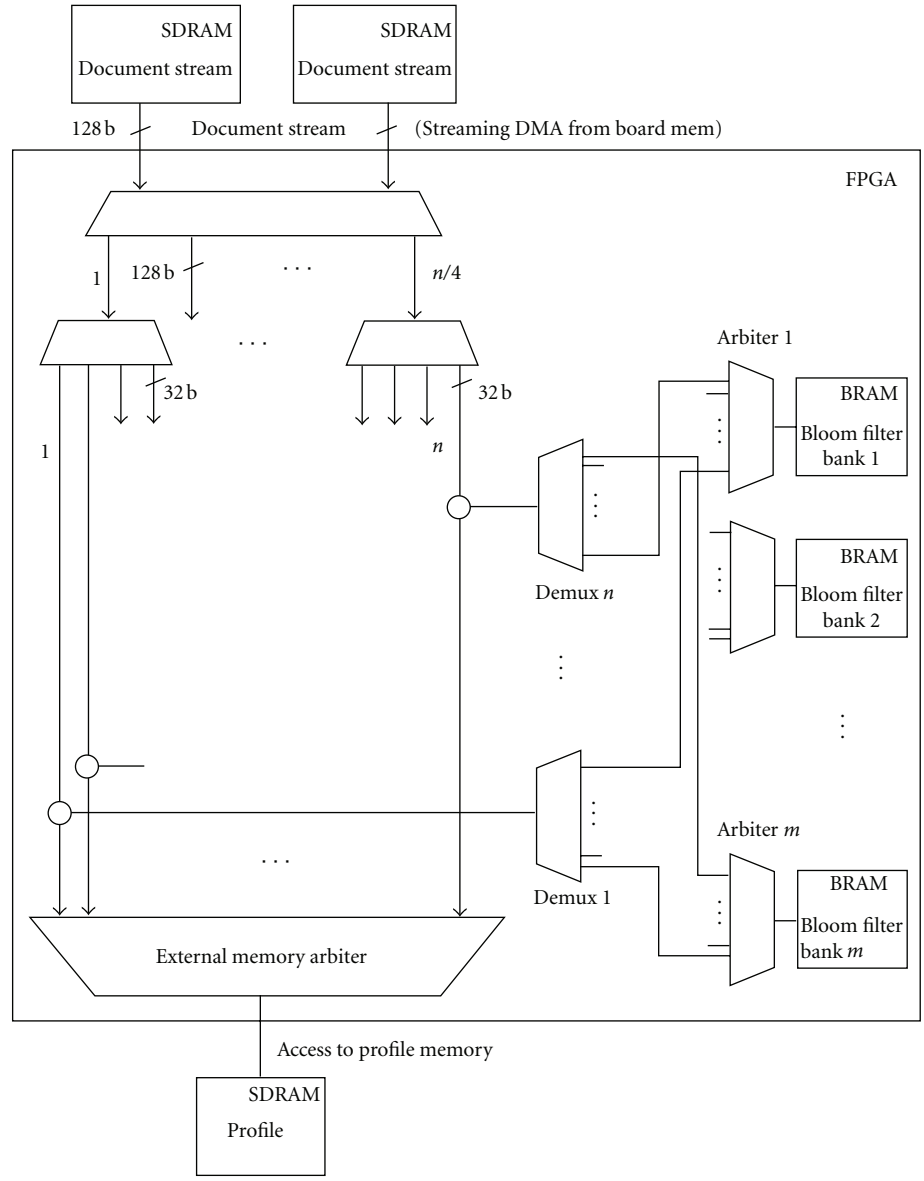



Figure 2: Parallelizing lookups using parallel streams and a multibank Bloom filter.

the time required for a single lookup in the Bloom filter. We also account for a fixed cost of contention $\Delta t_C$. We use a combinatorial approach; we count all possible arrangements of $n$ accesses to $m$ banks. Then we count the arrangements that result in $c$ concurrent accesses to a bank.

To do so, we need first to compute the *integer partitions* of $n$ [10] as they constitute all possible arrangements of $n$ accesses. For the remainder of the paper, we will refer to "all possible arrangements that result in $x$" as the *weight* of $x$. Each partition of $n$ will results in a particular average access time over all accesses. If we know the probability that each partition will occur and its resulting average access time, we can compute the total average access time.

*3.1.1. Integer Partitions.* A *partition* $p(n,k)$ of a positive integer $n$ is a nonincreasing sequence of $k$ positive integers $p_1, p_2, \ldots, p_k$ with $n$ as their sum. Each integer $p_i$ is called a *part*. Thus, with $n$ in our case being the number of access ports to our Bloom filter, each partition is a possible access pattern for the Bloom filter. For example, if $n = 16$ and $k = 8$, the partition (5 3 2 2 1 1 1 1) means that the first bank in the Bloom filter gets 5 concurrent accesses, the next 3, and so on. For $n \leq m$, $k \in [1,n]$; if $n > m$, we must restrict $k$ to $k \in [1,m]$ because we cannot have more than $m$ parts in the partition as $m$ is the number of banks. In other words, $k \in [1, \min(n,m)]$. We denote this as $p(n,k)$.

*3.1.2. Probability of Each Partition.* For each partition, we can compute the probability of it occurring as follows: if there are $n$ concurrent accesses to the Bloom filter's $m$ banks, $n \leq m$, then each access pattern can be written as a sequence of numbers. We are not interested in the actual numbers, but in

the patterns, for example, with $n = 8$ and $m = 16$, we could have a sequence $(a\ a\ a\ b\ b\ c\ c\ d)$, $a, b, c, d \in 0 \ldots m-1; a \neq b \neq c \neq d$ which results in a partition (3 2 2 1). Consequently, given a partition we need to compute the probability for the sequence which it represents. The probability for each number occurring is the same, $1/m$. We can compute this probability as a product of three terms. First, we consider the probabilities for sequences of length $n$ of events with probability $\alpha_i$ where each event occurs $x_i$ times. These are given by the multinomial distribution

$$n! \prod_{i=1}^{k} \frac{\alpha_i^{x_i}}{x_i!}, \tag{3}$$

where $0 < x_i \leq n$ and $n = \sum_{i=1}^{k} x_i$.

In our case, each event has the same probability $1/m$, and the number of times each event occurs is the size of each part $p_i$ in the partition, so

$$\frac{n!}{m^n} \prod_{i=1}^{k} \frac{1}{p_i!}. \tag{4}$$

This gives the probability for a sequence of $k$ groups of $p_i$ events, $n$ events in total.

The actual sequence will consist of numbers $1 \ldots m$, so we must consider the total number of different sequences of numbers that result in a given partition. This is simply the number of possible combinations of $k$ numbers out of $m$, $C_m^k$.

Finally, we must consider the permutations as wells for example, for (2 1 1) we must also consider (1 2 1) and (1 1 2). This is a combinatorial problem in which the bins are distinguishable by the number of elements they contain; however, the actual number of elements is irrelevant, only the fact that the bins are distinguishable. The derivation is slightly more complicated. We proceed as follows: we transform the partition into a tuple with as many elements as the number of different integers in the partition, and the value for each element is the number of times this integer occurs in the partition, for example, (5 5 3 3 2 1 1) $\rightarrow$ (2 2 1 2) and (4 3 2 2 1 1 1) $\rightarrow$ (1 1 2 3). We call the new set the frequencies of the partition $p$, $F(p(n,k))$. As partitions are nonincreasing sequences, the transformation is quite straightforward.

First we create an ordered set $\mathcal{S} = \{S_1, \ldots, S_i, \ldots\}$ with $P = \bigcup S_i$; that is, $\mathcal{S}$ is a set partition of $P$. The elements of $\mathcal{S}$ are defined recursively as

$$S_1 = \left\{ \forall p_j \in P \mid p_j = p_1 \right\}, \tag{5}$$

$$S_i = \left\{ \forall p_j \in P \setminus \bigcup_{k=1 \ldots i-1} S_k \mid p_j = p_1 \right\}. \tag{6}$$

That is, $S_1$ contains all parts of $P$ identical to the first part of $P$; for $S_2$, we remove all elements of $S_1$ from $P$ and repeat the process, and we continue recursively until the remaining set is empty. Finally, we create the (ordered) set of the cardinal numbers of all elements of $\mathcal{S}$

$$F = \left\{ f_i \triangleq S_i,\ \forall S_i \in \mathcal{S} \right\}. \tag{7}$$

We are looking for the permutations with repetition of $F(p(n,k))$, which is given by

$$n'! \prod_{\forall f_i \in F\left(p(n,k)\right)} \frac{1}{f_i!}, \tag{8}$$

where $n' = \sum f_i$.

Thus the final probability for each partition of $n$ and a given $m$ becomes

$$\mathcal{P}\left(p(n,k), m\right) = \frac{C_m^k}{m^n} \cdot n! \prod_{\forall p_i \in p(n,k)} \frac{1}{p_i!} \cdot n'! \prod_{\forall f_i \in F\left(p(n,k)\right)} \frac{1}{f_i!}. \tag{9}$$

We observe that

$$\sum_{k=1}^{n} \mathcal{P}\left(p(n,k), m\right) = 1 \tag{10}$$

regardless of the value of $m$.

In the next section we derive an expression for the access time for a given partition, depending on the number of accesses that will result in an external memory lookup.

*3.2. Average Access Time per Pattern.* The time to perform $n$ lookups in the Bloom filter is of course determined by the number of contending accesses. For $c$ contending accesses, it will take a time $c\Delta t_B$. However, not all Bloom filter lookups will result in a subsequent access to the external memory—in fact most of them will not, this is exactly the reason for having the Bloom filter. We will call a Bloom filter lookup that results in an access to the external memory a *hit*.

*3.2.1. Case of No Hits.* First, we will consider the case of 0 hits, that is, the most common case. In this case, the average access time for a given partition $p(n,k)$ is the average of all the parts in the partition

$$\overline{\Delta t_{H,p,0}} = \frac{k_{>1}}{k} \cdot \Delta t_C + \frac{n}{k} \Delta t_B, \tag{11}$$

where $k_{>1}$ is the number of parts $p_i > 1$. For the case of $k = n$ (no contention), $k_{>1} = 0$ so there is no fixed cost of contention $\Delta t_C$. Note again that $k \leq \min(n, m)$.

In practice, a small number of Bloom filter lookups will result in a hit, and consequently there is a chance of having one or more hits for concurrent accesses.

*3.2.2. Case of a Single Hit.* Consider the case of a single hit (out of $n$ lookups). The question we need to answer is, how long on average will it take to encounter a hit? Because as soon as we encounter a hit, we can proceed to perform the external memory access, without having to wait for subsequent hits. This time depends on the particular integer partition. To visualise the partition, we use a so-called Ferrers diagram [11], in which every part is arranged vertically as a list of dots. For example, consider the Ferrers diagram for the partition (8 4 1 1 1 1), that is, $n = 16$, $k = 6$. (Figure 3).

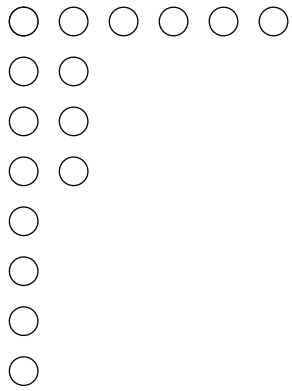


Figure 3: Ferrers diagram for the partition (8 4 1 1 1 1).

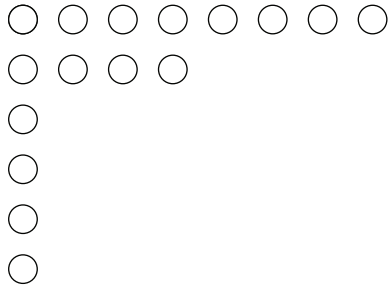


Figure 4: Ferrers diagram for the conjugate partition (6 2 2 2 1 1 1 1).

Each row can be interpreted as the number of concurrent accesses to different banks; each column represents the number of contending accesses to a particular bank.

From this graph it is clear that the probability for finding the hit on the first cycle is 6/16; on the second to fourth cycle 2/16, on the fifth to eighth cycle 1/16. Consequently, the average time to encounter a hit will in this case be

$$1 \cdot \frac{6}{16} + (2+3+4) \cdot \frac{2}{16} + (5+6+7+8) \cdot \frac{1}{16}. \tag{12}$$

To generalise this derivation, we observe first that the transposition of the Ferrers diagram of an integer partition $p$ yields a new integer partition $p'(n,k')$ for the same integer called the *conjugate* partition. In our example $p' = (6\ 2\ 2\ 2\ 1\ 1\ 1\ 1)$ with $k' = 8$ (Figure 4).

We observe that the time it takes to reach a hit in part $p'_i$ is $\Delta t_B \cdot i$. Using the conjugate partition $p'$, we can write the lower bound for average time it takes to reach a hit in partition $p$ as

$$\overline{\Delta t_{H,p,1}} = \left(1 - \frac{k - k_{>1}}{n}\right) \cdot \Delta t_C + \Delta t_B \cdot \frac{1}{n} \sum_{i=1}^{k'} i \cdot p'_i. \tag{13}$$

The term in $\Delta t_C$ only occurs when the hit is in a bank with contention, that is, in a part greater than 1. There are $k - k_{>1}$ parts of size 1, so the chance of a hit occurring in one of them (i.e., a hit on a bank without contention) is $k - k_{>1}/n$. Thus, the probability for the term in $\Delta t_C$ is

$$1 - \frac{k - k_{>1}}{n}. \tag{14}$$

And of course, as the hit results in an external access, the average access time is

$$\overline{\Delta t_{A,p,1}} = \left(1 - \frac{k - k_{>1}}{n}\right) \cdot \Delta t_C + \Delta t_B \cdot \frac{1}{n} \sum_{i=1}^{k'} i \cdot p'_i + \frac{1}{n} \cdot \Delta t_S. \tag{15}$$

For the case of $k = n$, the equation reduces to

$$\overline{\Delta t_{A,p,1}} = \Delta t_B + \frac{1}{n} \cdot \Delta t_S. \tag{16}$$

*3.2.3. Case of Two or More Hits.* If there are two or more hits, the exact derivation would require enumerating all possible ways of distributing $n_H$ hits over a given partition; furthermore, simply enumerating them is not sufficient; we would have to consider the exact time of occurrence of each hit to be able to determine if a subsequent hit was encountered during or after the time to perform an external lookup and compute the score ($\Delta t_S$) from a given hit. It is easy to see that, for large $n_H$, this problem is so complex as to be intractable in practice. However, we can make a simplifying assumption; in practice, $\Delta t_S$ will be much larger than the time to perform a Bloom filter lookup.

If that is the case, a good approximation for the total elapsed time is the time until the *first* hit is encountered plus $n_H$ times the time for external access. This approximation is exact as long as the time it takes to sequentially perform all external lookups is longer than the time between the best and worst case Bloom filter access time for $n_H$ hits on a single bank, in other words as long as $\Delta t_S > p_i \Delta t_B$. The worst case is of course $p_1 = n$, but this case has a very low probability; for example, for $n = 16$, the average value of all parts is 2.5; even considering only the parts >1, the average is still <4. For $n = 32$, the numbers are, respectively, 3 and 5. In practice, if $\Delta t_S / \Delta t_B > 10$, the error will be negligible.

Conversely, we could consider the time until the *last* hit is encountered plus $n_H$ times $\Delta t_S$. This approximation provides an upper bound for the access time.

Therefore, we are only interested in these two cases, that is, the lowest, respectively, highest part of the partition with at least one hit. We need to compute the probability that the lowest (resp. highest) part will contain a hit, and the next but lowest (resp. highest) one, and so forth. For simplicity, we leave off $\Delta t_C$ in the following derivation.

*Lower Bound.* The number of all possible cases is $N_{p'} = C(n, n_H)$, all possible arrangements of $n_H$ elements in $n$ bins. To compute the weight of a hit in the lowest part $p'_1$, we compute the complement, all possible arrangements without any hits in $p'_1$. That means that we remove $p'_1$ from $n$. Then, using the notation $\neg p_1$ for "not a hit in $p'_1$," we compute

$$N_{\neg p_1} = C(n - p'_1, n_H). \tag{17}$$

These are all the possible cases for not having a hit in $p'_1$. Thus, $N_{p_1} = N_p - N_{\neg p_1}$ is the number of possible arrangements with $1 \dots n_H$ hits in $p'_1$.

We now do the same for $p_2$, and so forth. That gives us all possible cases for *not* having a hit in $p_i$

$$N_{\neg p_i} = C\left(n - \sum_{j=1}^{i} p'_j, n_H\right). \tag{18}$$

Obviously, there must be enough space in the remaining parts to accommodate $n_H$ hits, so $i$ is restricted to values where

$$n - \sum p'_i \geq n_H. \tag{19}$$

We call the highest index for which (19) holds, $k^*$.

To obtain the weight of a hit in $p'_i$, we must of course subtract the weight of a hit in $p'_{i-1}$, because $N_p - N_{\neg p_i}$ would give the weight for having a hit in all parts up to $p_i$. It is easy to show (by substitution of (17)) that

$$N_{p_i} = N_{\neg p_{i-1}} - N_{\neg p_i}. \tag{20}$$

Finally, the average time it takes to reach a part in a given $p'$ with at least one hits out of $n_H$ is

$$\overline{\Delta t_{H,p,n_H}} = \Delta t_B \cdot \frac{1}{N_{p'}} \sum_{i=1}^{k^*} i \cdot N_{p_i}. \tag{21}$$

With the above assumption, the average access time for $n_H$ hits can then be approximated as

$$\overline{\Delta t_{A,p,n_H}} = \left(1 - \frac{k - k_{>1}}{n}\right) \cdot \Delta t_C + \Delta t_B \cdot \frac{1}{N_{p'}} \sum_{i=1}^{k^*} i \cdot N_{p_i} + n_H \Delta t_S. \tag{22}$$

We observe that for $n_H = 1$, (22) indeed reduces to (15) as $N_{p'} = n$ and $N_{p_i} = p'_i$. For $n = k$ the equation reduces to $\Delta t_B + n_H \Delta t_S$.

*Upper Bound.* The upper bound is given by the probability that the highest part is occupied, and so forth, so the formula is the same as (18) but starting from the highest part $p_c$, that is,

$$N_{\neg p_{c-i}} = C\left(n - \sum_{j=c-i+1}^{c} p'_j, n_H\right) \tag{23}$$

with the corresponding restriction on $i$ that

$$\sum_{i=1}^{k^*} p'_i \geq n_H. \tag{24}$$

As we will see in Section 3.5, in practice the bounds are usually so close together that the difference is negligible.

*3.3. Probability of External Memory Access.* The chance that a term will occur in the profile depends on the size of the profile $N_{\mathcal{P}}$ and the size of the vocabulary $N_{\mathcal{V}}$

$$P_{\mathcal{P}} = \frac{N_{\mathcal{P}}}{N_{\mathcal{V}}}. \tag{25}$$

This is actually a simplified view; it assumes that the terms occurring in the profile and the documents are drawn from the vocabulary in a uniform random way. In reality, the probability depends on how discriminating the profile is. As the aim of a search is of course to retrieve only the relevant documents, we can assume that actual profiles will be more discriminating than the random case. In that case (25) provides a worst case estimate of contention.

The probability of $n_H$ hits, that is, contention between $n_H$ accesses to the external memory is then

$$C(n, n_H) \cdot P_{\mathcal{P}}^{n_H} \cdot (1 - P_{\mathcal{P}})^{n - n_H}. \tag{26}$$

That is, there are $C(n, n_H)$ arrangements of $n_H$ accesses out of $n$, and, for each of them, the probability that it occurs is $P_{\mathcal{P}}^{n_H} \cdot (1 - P_{\mathcal{P}})^{n-n_H}$. Furthermore, $n_H$ contending accesses will take a time $n_H \Delta t_S$. Of course, if no external access is made, the external access time is 0.

*3.4. Average Overall Throughput*

*3.4.1. Average Access Time over All $n_H$ for a Given Pattern.* We can now compute the average access time over all $n_H$ for a given access pattern $p$ by combining (22) and(26)

$$\overline{\Delta t_{A,p}} = \sum_{n_H=0}^{n} C(n, n_H) \cdot P_{\mathcal{P}}^{n_H} \cdot (1 - P_{\mathcal{P}})^{n-n_H} \overline{\Delta t_{A,p,n_H}}. \tag{27}$$

*3.4.2. Average Access Time over All Patterns for Given $n$ and $m$.* Finally, using (9) and (27), we can compute the average access time over all patterns for given $n$ and $m$, that is, the average overall throughput of the application with $n$ parallel threads and an $m$-bank Bloom filter

$$\overline{\Delta t_A(n, m)} = \sum_{\forall p(n)} \mathcal{P}(p(n, k), m) \cdot \overline{\Delta t_{A,p}}. \tag{28}$$

*3.5. Analysis.* In this section the expression obtained in Section 3.4 is used to investigate the performance of the system and the impact of the values of $n$, $m$, $\Delta t_B$, $\Delta t_S$, and $P_{\mathcal{P}}$ on the throughput.

*3.5.1. Accuracy of Approximation.* To evaluate the accuracy of the approximations introduced in Section 3.2.3, we compute the relative difference between the "first hit" approximation and the "upper bound" approximation. From Figure 5, it can be seen that the difference is less than 1% of the throughput over all simulated cases. As the upper bound always overestimates the delay, and the "first hit" approximation will in most cases return the correct delay, this demonstrates that both approximations are very accurate. An interesting observation is that for $\Delta t_B = 4$ the error is almost the same as

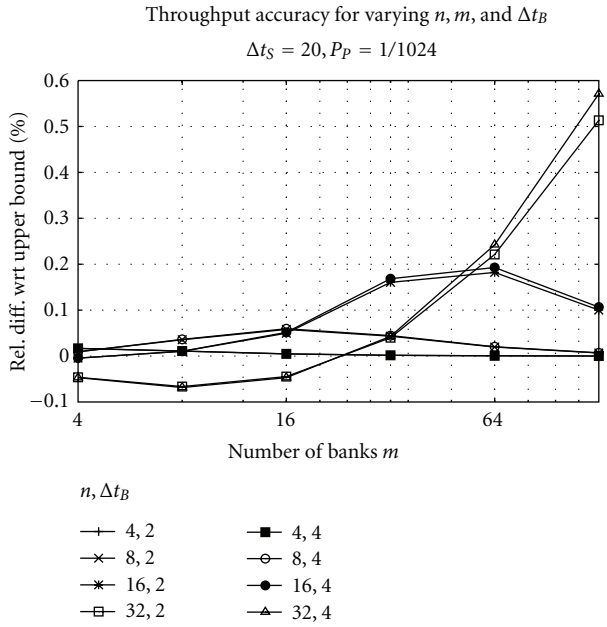



FIGURE 5: Accuracy of the approximation for $n_H \geq 2$.

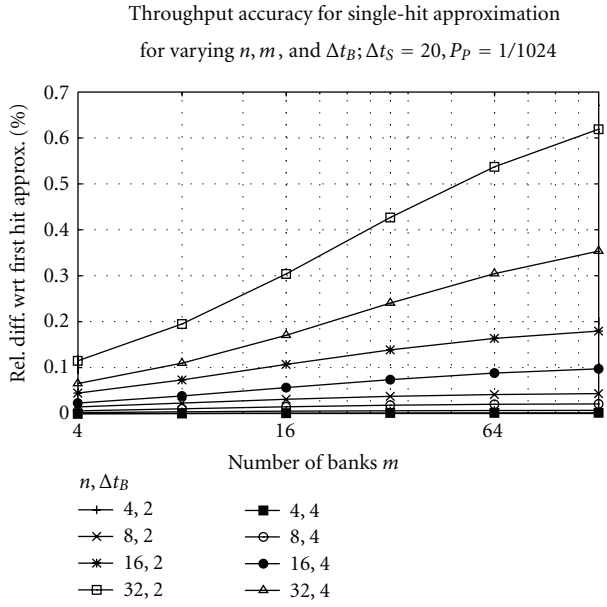



FIGURE 6: Accuracy of the single-hit approximation for $n_H \geq 2$.

for $\Delta t_B = 2$, which illustrates that the condition $\Delta t_S > p_i \Delta t_B$ is sufficient but not necessary.

Next, we consider a more radical approximation; we assume that, for $n_H > 1$, $P_{\mathcal{P}} = 0$, in other words we ignore all cases with more than 1 hit.

From Figure 6 we see that the relative difference between the throughput using this approximation and the "first hit" is very small, to such an extent that in almost all cases it is justified to ignore $n_H > 1$. This is a very useful result as this approximation speeds up the computations considerably.

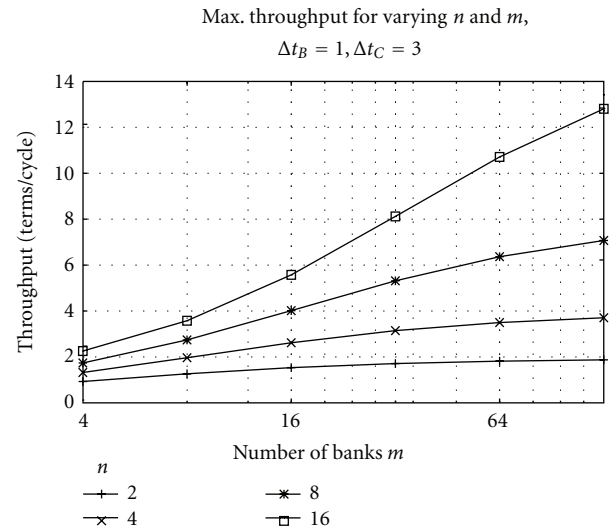



FIGURE 7: Best case (0 hits) average access time for a Bloom filter with $m$ banks and $n$ access ports, $\Delta t_B = 1$, $\Delta t_C = 3$.

*3.5.2. Maximum Achievable Throughput.* The throughput depends on the number of hits in the Bloom filter. Let us consider the case where the Bloom filter contains no hits at all. This is the maximum throughput the system could achieve, and it corresponds to a profile for which no document in the stream has any matches. We can use (11) and (9) to calculate the best-case average access time for a Bloom filter with $m$ banks and $n$ access ports

$$\overline{\Delta t_{\min}(n,m)} = \sum_{k=1}^{n} \sum_{\forall p(n,k)} \left( \left( \frac{k_{>1}}{k} \Delta t_C + \frac{n}{k} \Delta t_B \right) \cdot \mathcal{P}(p(n,k),m) \right). \tag{29}$$

Note that for $m < n$, $\mathcal{P}(p(n,n)) = 0$.

The results are shown in Figure 7. The figure shows that for $\Delta t_B = 1$, $\Delta t_C = 3$ (the values for our current implementation), the I/O-limited throughput (4 terms/cycle for the PROCStar-III board) is achieved with $n = 8$ and $m = 16$. That means that we need to demultiplex both input streams into 4 parallel streams because each 64-bit word contains 2 terms.

*3.5.3. Throughput Including External Access.* Figure 8 shows the effect of the external memory access and score computation. The important observation is that the performance degradation is quite small for low hit rates, and still only around 25% for a relatively high hit rate of 1/512. This demonstrates that the assumptions underlying our design are justified.

*3.5.4. Impact of Bloom Filter Access Time.* A further illustration of the impact of $\Delta t_B$ is given in Figure 9, which plots the throughput as a function of $\Delta t_B$ on a log/log scale. This figure illustrates clearly how a reduction in throughput as result of slower Bloom filter access can be compensated for by increasing the number of access streams. Still, with $\Delta t_B = 4$,

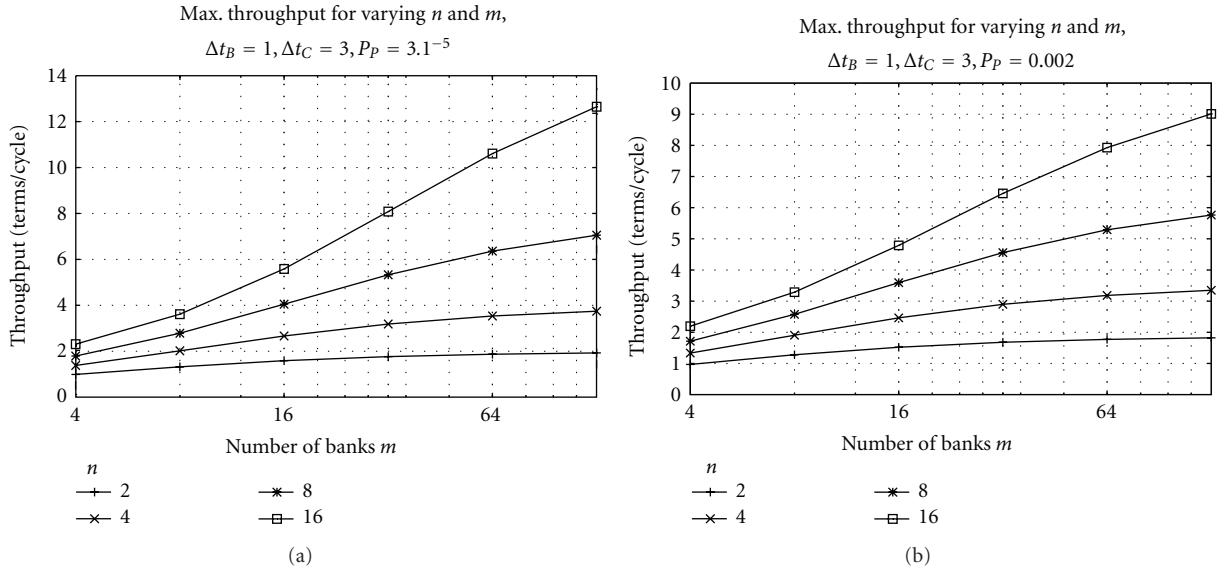



Figure 8: Average access time for a Bloom filter with $m$ banks and $n$ access ports, $\Delta t_S = 20$, (a) $P_{\mathcal{P}} = 3 \cdot 10^{-5}$ (b) $P_{\mathcal{P}} = 0.002$.

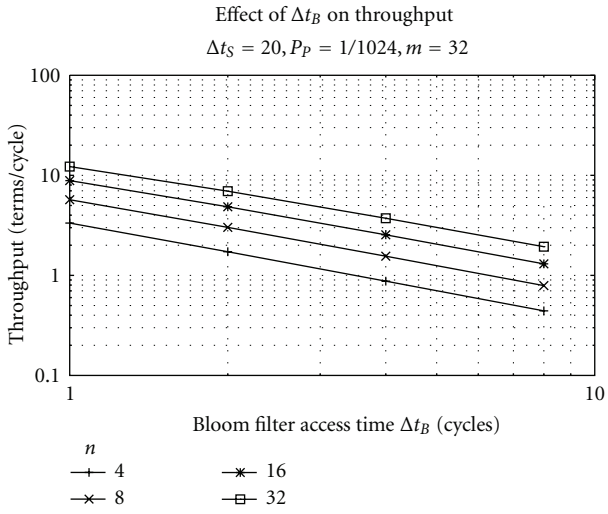



Figure 9: Impact of Bloom filter access time on throughput.

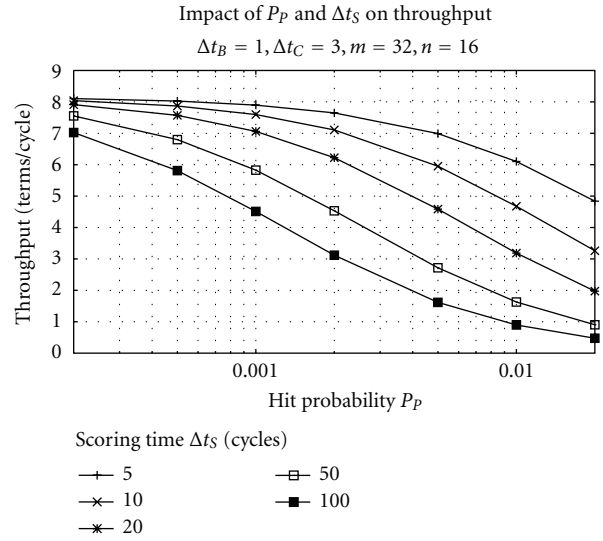



Figure 10: Impact on throughput of hit probability and external memory access time.

we would need 32 parallel streams per input stream, or we would need a very large number ($\gg 128$) Bloom filter banks. On the one hand, the upper limit is 512 (the number of M9K blocks on the Stratix-III 260E FPGA); on the other hand, the size of the demultiplexers and arbiters would become prohibitive as it grows as $m \cdot n$.

*3.5.5. Impact of Profile Hit Probability and External Memory Access Time.* The final figure (Figure 10) is probably the most interesting one. It shows how, for very selective profiles (i.e., profiles resulting in very low hit rates), the effect of long external memory access times is very small.

## 4. FPGA Implementation

We implemented our design on the GiDEL PROCStar-III development board (Figure 11). This system provides an extensible high-capacity FPGA platform with the GiDEL PROC-API library-based developer kit for interfacing with the FPGA.

*4.1. Hardware.* Each board contains four Altera Stratix-III 260 E FPGAs running at 125 MHz. Each FPGA supports a five-level memory structure, with three kinds of memory blocks embedded in the FPGA:

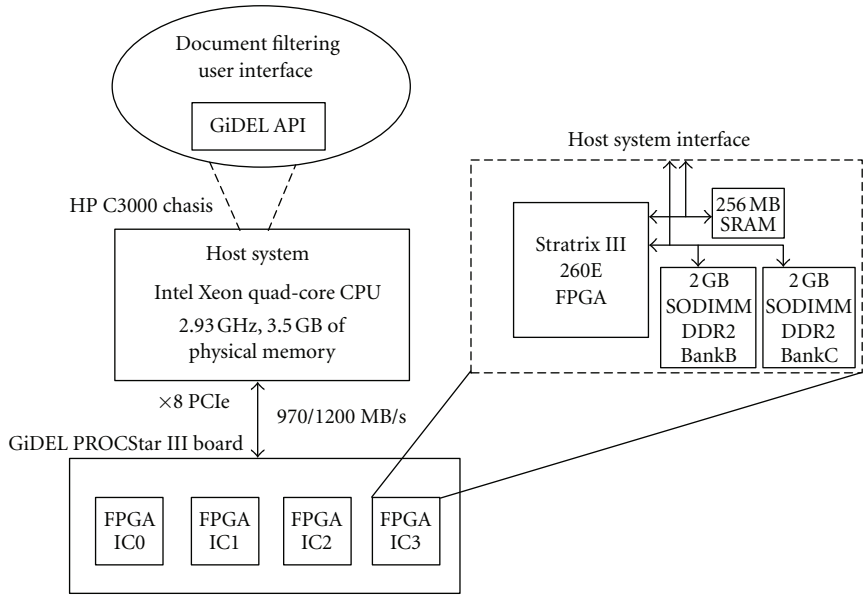



Figure 11: Block diagram of FPGA platform and photograph of experimental hardware.

(i) 5,100 MLAB RAM blocks (320 bit),

(ii) 864 M9K RAM blocks (9 Kbit), and

(iii) 48 M144K blocks (144 Kbit)

and 2 kinds of external DRAM memory:

(i) 256 MB DDR2 SDRAM onboard memory (Bank A) and

(ii) two 2 GB SODIMM DDR2 DRAM memories (Bank B and Bank C).

The embedded FPGA memories run at a maximum frequency of 300 MHz, Bank A and Bank B at 667 MHz, and Bank C at 360 MHz. The FPGA-board is connected to the host platform via 8-lane PCI Express I/O interface. The host system consists of a quad-core 64-bit Intel Xeon X5570 CPU with a clock frequency of 2.93 GHz and 3.5 GB DDR2 DRAM memory, the operating system is 32-bit Windows XP. The host computer transfers data to the FPGA using 32-bit DMA channels.

*4.2. Development Environment.* FPGA-accelerated applications for the PROCStar board are implemented in C++ using the GiDEL PROC-API libraries for interacting with the FPGA. This API defines a hardware abstraction layer that provides control over each hardware element in the system; for example, Memory I/O is implemented using the GiDEL MultiFIFO and MultiPort IPs. To achieve optimal performance, we implemented the FPGA algorithm in VHDL (as opposed to Mitrion-C as used in our previous work). We used the Altera Quartus toolchain to create the bitstream for the Stratix-III.

*4.3. FPGA Implementation Description.* Figure 12 presents the overall workflow of our implementation. The input stream of document term pairs is read from the SDRAM via a FIFO. A Bloom filter is used to discard negatives (terms that do not appear in the profile) for multiple terms in parallel. Profile weights are read corresponding to the positives, and the scores are computed for each term in parallel and accumulated to achieve the final score described in (1). Below, we describe the key modules for the implementation: document streaming, profile negative hit filtering, and profile lookup and scoring.

*4.3.1. Document Streaming.* Using a bag-of-words representation (see Section 2) for the document, the document stream is a list of *(document id, document term tuple set)* pairs. The FPGA accepts a stream of 64-bit words from the 2 GB DRAM (Bank B). Consequently, the document stream must be encoded onto this word stream. The document term tuple $d_i = (t_i, f_i)$ can be encoded in 32 bits: 24 bits for the term id (supporting a vocabulary of 16 million terms) and 8 bits for the term frequency. Thus, we can combine two tuples into a 64-bit word. To mark the start and end of a document, we insert a marker words (64 bits) followed by the document id (64 bits).

*4.3.2. Profile Negative Hit Filtering.* As described in Section 2, we implemented a Bloom filter in the FPGA's on-chip MRAM (M9K blocks). The higher internal bandwidth of the MRAMs leads to very fast rejection of negatives. Although the MRAM is fast, concurrent lookups lead to contention. To reduce contention we designed a distributed Bloom filter. Based on the analysis presented in this paper, the Bloom filter memory is distributed over a large number of banks (16 in the current design) and a crossbar switch connects the document terms streams to the banks. As shown in our analysis, in this way contention is significantly reduced. The design was implemented as shown in Figure 2, but due to

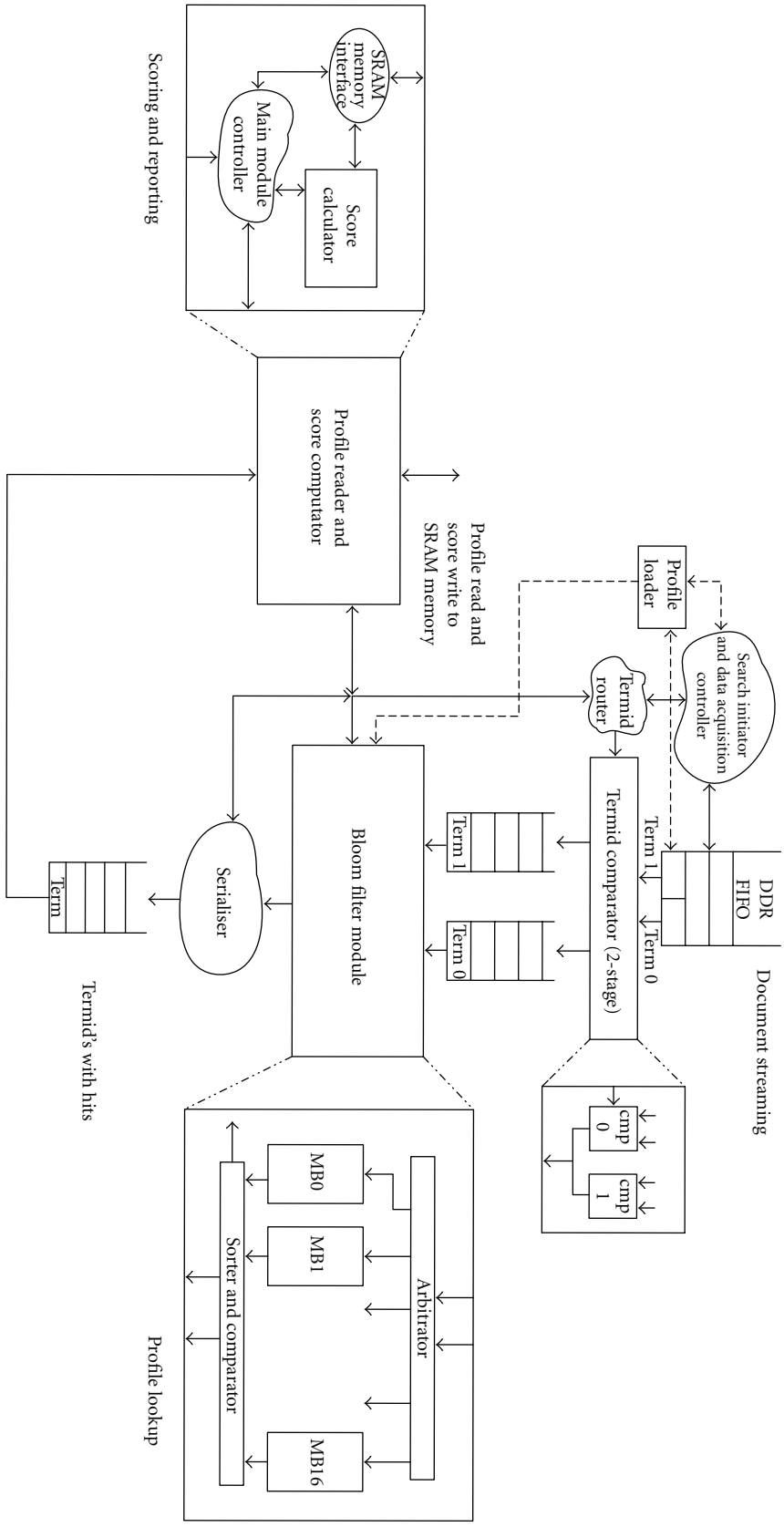



Figure 12: Overall block diagram of FPGA implementation.

an issue with the board we could only use one SDRAM to store the collection. As a result we have only two parallel terms in the current implementation.

*4.3.3. Profile Lookup and Scoring.* As explained in Section 2.2, the actual lookup and scoring system is quite straightforward; the input stream is scanned for header and footer words. The header word action is to store the subsequent document ID and to set the corresponding document score to 0; the footer word action is to collect and output the *(document ID, document score)* pair if the score exceeds the threshold. For every two terms in the document, first the Bloom filter is used to discard negatives, and then the weights corresponding to positives are read from the SDRAM. The score is computed for each of the terms in parallel and added. The score is accumulated for all terms in the document, and finally the score stream is filtered against a limit before being output to the host. Figure 13 summarises the implementation of the profile lookup and scoring.

*4.3.4. Discussion.* The implementation above leverages the advantages of an FPGA-based design, in particular the memory architecture of the FPGA; on a general-purpose CPU-based system, it is not possible to create a very fast, very low-contention Bloom filter to discard negatives. Also, a general-purpose CPU-based system only has a single, shared memory. Consequently, reading the document stream will contend for memory access with reading the profile terms, and as there is no Bloom filter, we have to look up each profile term. We could of course implement a Bloom filter, but, as it will be stored in main memory as well, there is no benefit; looking up a bit in the Bloom filter is as costly as looking up the term directly. Furthermore, the FPGA design allows for lookup and scoring of several terms in parallel.

*4.4. FPGA Utilisation Details.* Our implementation used only 11,033 of the 203,520 logic elements (LEs) or a 5% utilisation of the logic in the FPGA, and 4,579,824 out of 15,040,512 for a 30% utilisation of the RAM. Of the 11,033 LEs utilised by whole design on the FPGA, the actual document filtering algorithm only occupied 1,655 LEs, which is less than 1% of utilisation, and rest was used by the GiDEL Memory IPs. The memory utilised for the whole design (4,579,824 bits) was mainly for the Bloom filter that is mapped on embedded memory blocks (MRAMs). The Quartus PowerPlay Analyzer tool estimates the power consumption of the design to be 6 W. The largest contribution to the power consumption is from the memory I/O.

## 5. Evaluation

In this section we discuss our evaluation results. We present our experimental methodology and the data summarising the performance of our FPGA evaluation and comparison with non-FPGA-accelerated baselines, and we conclude with the learnings from our experiments.

Table 1: Summary statistics from representative real-world collections that we used as templates for our synthetic data sets.

| Collection | No. docs. | Avg. Doc. Len. | Avg. Uniq. Terms |
|---|---|---|---|
| Aquaint | 1,033,461 | 437 | 169 |
| USPTO | 1,406,200 | 1718 | 353 |
| EPO | 989,507 | 3863 | 705 |

*5.1. Creating Synthetic Data Sets.* To accurately assess the performance of our FPGA implementation, we need to exercise the system on real-world input data; however, it is hard to get access to such real-world data; large collections such as patents are not freely available and governed by licenses that restrict their use. For example, although the researchers at Glasgow University have access to the TREC Aquaint collection and a large patent corpus, they are not allowed to share these with a third party. In this paper, therefore, we use synthetic document collections statistically matched to real-world collections. Our approach is to leverage summary information about representative datasets to create corresponding language models for the distribution of terms and the lengths of documents; we then use these language models to create synthetic datasets that are statistically identical to the original data sets. In addition to addressing IP issues, synthetic document collections have the advantages of being fast to generate and easy to experiment with, and not taking up large amounts of disk space.

*5.1.1. Real-World Document Collections.* We analysed the characteristics of several document collections—a newspaper collection (TREC Aquaint) and two collections of patents from the US Patent Office (USPTO) and the European Patent Office (EPO). These collections provide good coverage on the impact of different document lengths and sizes of documents on filtering time. We used the Lemur (http://www.lemurproject.org/) Information Retrieval toolkit to determine the rank frequency distribution for all the terms in the collection. Table 1 shows the summary data from the collections we studied as templates.

*5.1.2. Term Distribution.* It is well known (see, e.g., [12]) that the rank-frequency distribution for natural language documents is approximately Zipfian

$$f(k;s;N) = \frac{1/k^s}{\sum_{n=1}^{N} 1/n^s}, \tag{30}$$

where $f$ is frequency of term with rank $k$ in randomly chosen text of natural language, $N$ is number of terms in the collection, and $s$ is an empirical constant. If $s > 1$, the series becomes a value of a Riemann $\zeta$-function and will therefore converge. This type of distribution approximates a straight line on a log-log scale. Consequently, it is easy to match this distribution to real-world data with linear regression.

Special purpose texts (scientific articles, technical instructions, etc.) follow variants of this distribution. Montemurro [13] has proposed an extension to Zipf's law which better captures the linguistic properties of such collections.

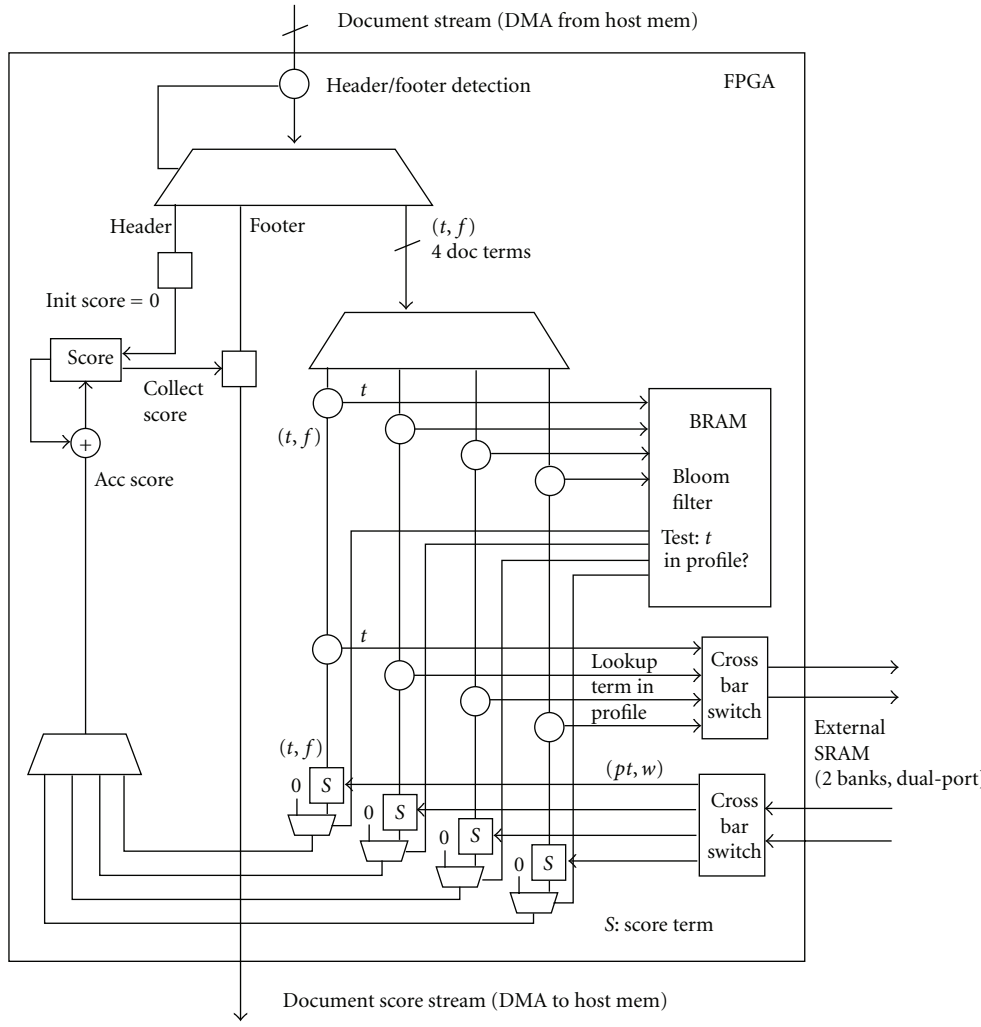



Figure 13: Implementing profile lookup and scoring.

His proposal is based on observation that, in general after some pivot point $p$, the probability of finding a word of rank $r$ in the text starts to decay much faster than in the beginning. In other words, in log-log scale, the low-frequency part of the distribution has a steeper slope than the high-frequency part. Consequently, the distribution can be divided into two regions each obeying the power law, but with different slopes

$$F(r) = \begin{cases} a_1 r + b_1 & r < p, \\ a_2 r + b_2 & \text{otherwise}, \end{cases} \tag{31}$$

We determine the coefficients $a_1, a_2, b_1$, and $b_2$ from curve-fitting on the summary statistics from the real-world data collections. Specifically, we use the sum of absolute errors as the merit function combined with a binary search to obtain the pivot. We then use a least-squares linear regression, with $\chi^2$ statistics as a measure of quality (taken from [14]). A final normalisation step is added to ensure that the piecewise linear approximation is a proper probability density function.

*5.1.3. Document Length.* Document lengths are sampled from a truncated Gaussian. The hypothesis that the document lengths in our template collections have a normal distribution was verified using a $\chi^2$ test with 95% confidence. The sampled values are truncated at the observed minimum and maximum lengths in the template collection.

Once the models for the distribution of terms and document lengths are determined, we use these models to create synthetic documents of varying lengths. Within each document, we create terms that follow the fitted rank-frequency distribution. Finally, we convert the documents into the standard bag-of-words representation, that is, a set of unordered *(term, frequency)* pairs.

*5.2. Experimental Parameters.* Statistically, the synthetic collection will have the same rank-frequency distribution for the terms as the original data sets. Consequently, the probability that a term in the collection matches a term in the profile will be the same in the synthetic collection and the original

collection. The performance of the algorithm on the system now depends on

(i) the size of the collection,

(ii) the size of the profile,

(iii) the "hit probability," that is, the probability that the profile corresponding to a term has a nonzero weight.

To evaluate these effects, we studied a number of different configurations—with different document sizes, different profile lengths, and different profile constructions. Specifically, we studied profile sizes of 4 K, 16 K, and 64 K terms, the first two are of the same order of magnitude as the profile sizes for TREC Aquaint and EPO as used in our previous work [2], and the third, larger profile was added to investigate the impact of the profile size. We studied two different document collections: 128 K documents of 2048 terms, which is representative for the patent collections, and 512 K documents of 512 terms, similar to the Aquaint collection. Note that the total size of the collection is not important for the performance evaluation; for both the CPU and FPGA implementation, the time taken to filter a collection is proportional to its size.

We evaluated four ways of creating profiles. The first way ("Random") is by selecting a number of random documents from the collection until the desired profile size is reached. These documents were then used to construct a relevance model. The relevance model defined the profiles which each document in the collection was matched against (as if it were being streamed from the network). The second type of profiles ("Selected") was obtained by selecting terms that occur in very few documents (less than ten in a million). For our performance evaluation purpose, the main difference between these profiles is the hit probability, which was $10^{-5}$ for the "Random" profiles and $5 \cdot 10^{-4}$ for the "Selected" profiles. For reference, we also compared the performance against an "Empty" profile (one that results in no hits).

### 5.3. FPGA Performance Results

*5.3.1. Access Time Measurements.* The performance of the FPGA was measured using a cycle counter. The latency between starting the FPGA and the first term score is 22 cycles. For the subsequent terms, the delay depends on a number of factors. We considered three different cases:

(i) "Best Case": no contention on the Bloom filter access and no external memory access

(ii) "Bloom Filter Contention": contention on the Bloom filter access for every term but no external memory access

(iii) "External Access": no contention on the Bloom filter access, external memory access for every term

These cases were obtained by creating documents with contending/not contending term pairs and by setting all Bloom filter bits to 0 (no external access, which corresponds to an empty profile) or 1 (which correspond to a profile that would contain all terms in the vocabulary).

Table 2: FPGA Cycle counts for different cases.

| Case | No. cycles/2 terms | Probability |
|---|---|---|
| Best case | 1 | .9375 |
| Bloom filter contention | 5 | .0625 |
| External access | 37 | <0.00001 |

The results are shown in Table 2. As we read two terms in parallel, the Best Case (i.e., the case of no contention and no hits) demonstrates that the FPGA implementation does indeed work at I/O rates, that is, $\Delta t_B = 1$. The table also shows the probability for each case when filtering an actual document collection.

The most interesting result in Table 2 is the "Bloom Filter Contention," which shows that in our design $\Delta t_C = 3$. The case of "External Access," which means no contention on the Bloom filter, and lookup of both terms in the external memory shows that $\Delta t_S = 18$.

As explained in Section 3, the Bloom filter contention depends on the number of Bloom filter banks (2 parallel terms, 16 banks), and, in the current design, the contention probability is 1/16 (from (9)). The probability for external access depends on the actual document collection and profile, but as the purpose of a document filter is to retrieve a small set of highly relevant documents, this probability is typically very small, as demonstrated by the experiments discussed in the next section. Consequently, the typical performance is determined by the cycle counts for Best Case and Bloom Filter Contention. Using (29), we get $(1/2)\Delta t_B \cdot (15/16) + ((1/2)\Delta t_C + \Delta t_B) \cdot (1/16) = 0.625$ cycles per term. At a clock speed of 125 MHz, this results in a throughput of 200 million terms per second (200 MT/s) per FPGA.

*5.3.2. Comparison with CPU Reference Systems.* Table 3 presents performance results for our FPGA implementation for various workload types. Focusing on a "Random" profile of 16 K terms for 128 K documents, our measured performance is close to 800 million terms/second for the design—close to the estimated performance; the earlier calculations showed 200 million terms/second *per FPGA*: across the four FPGAs in the GiDEL board, that translates to 800 million terms/second for the design. Table 3 also shows the sensitivity to various other parameters. The performance of the FPGA design is comparable for different profile sizes and document sizes. However, as expected, the performance varies based on different hit probabilities for different profiles.

To compare the FPGA performance against a conventional CPU, we ran the experiments discussed in Section 5.1 on an optimised reference implementation (compared to the Lemur-based implementation used in our previous work), written in C++, compiled with g++ with optimisation -O3, and run on two different platforms: *System1* (an iMac) has an Intel Core 2 Duo Mobile E8435 CPU with clock frequency 3.06 GHz and 8 GB RAM, bus speed 1067 MHz. *System2* (a Linux server) has an Intel Core i7-2600 CPU running at 3.4 GHz, with 16 GB RAM, bus speed 1333 MHz. The higher memory configurations are required to enable sufficient

Table 3: Throughput of document filtering application (M terms/s) for (a) 128 K documents of 2048 terms and (b) 512 K documents of 512 terms.

(a)

| Profile | System1 | System2 | FPGA board |
|---|---|---|---|
| Empty, 4 K | 31 | 48 | 800 |
| Empty, 16 K | 31 | 48 | 800 |
| Empty, 64 K | 31 | 48 | 800 |
| Random, 4 K | 25 | 42 | 800 |
| Random, 16 K | 24 | 41 | 800 |
| Random, 64 K | 24 | 41 | 800 |
| Selected, 4 K | 21 | 37 | 792 |
| Selected, 16 K | 18 | 35 | 792 |
| Selected, 64 K | 18 | 25 | 792 |

(b)

| Profile | System1 | System2 | FPGA board |
|---|---|---|---|
| Empty, 4 K | 30 | 53 | 800 |
| Empty, 16 K | 32 | 53 | 800 |
| Empty, 64 K | 32 | 53 | 800 |
| Random, 4 K | 26 | 47 | 800 |
| Random, 16 K | 26 | 46 | 800 |
| Random, 64 K | 25 | 46 | 800 |
| Selected, 4 K | 20 | 40 | 796 |
| Selected, 16 K | 19 | 38 | 792 |
| Selected, 64 K | 17 | 27 | 796 |

memory for the algorithm; it is not possible to run the reference implementation on the 32-bit Windows XP server which hosts the FPGA board as the 3.5 GB of memory is not sufficient. We could of course run the algorithm several times on smaller data sets but then in that case the time required to read the data from disk would dominate the performance. We keep the entire data set in memory because the memory I/O is much higher than the disk I/O. While this approach might not be practical on a CPU-based system, on the FPGA-based system this is entirely practical as the PROCStar-III board has a memory capacity of 32 GB. This means that, for example, the Novo-G FPGA supercomputer, which hosts 48 PROCStar-III boards, can support a collection of 1.5 TB. Note also that the format in which the documents are stored on the disk is a very efficient bag-of-words representation, which is much smaller than the actual textual representation of the document.

The results are summarised in Table 3. For example, focusing on one example case, for the random profile with 16 K terms and 128 K documents, compared to the 800 million terms/second performance per FPGA achieved by our design, the *System2* system achieves 41 million terms/second, and the *System1* system achieves 24 million terms/sec. This translates to a 36-fold speedup for the FPGA-based design relative to the *System1* system and a 20-fold speedup relative to the *System2* system. Additionally, examining the results for various workload configurations, the FPGA's performance is relatively constant across different workload inputs. This bears out the rationale for our design because in general hits are rare, the FPGA works at the speed determined by I/O and Bloom Filter performance. Unlike the FPGA-based design, the CPU-based system sees more variation in performance with profile size (degraded performance with increased profile size) and document size (degraded performance with larger documents) and a bigger dropoff in performance between various profile types compared to the FPGA-based design.

## 6. Discussion

In the above sections we have used a preliminary implementation of our proposed design to validate the analytical model. The design does indeed behave in line with the model, for the case of two parallel terms and a 16-bank Bloom filter. The performance is 200 M terms/s. This design is not optimal for several reasons. On the one hand, the original aim was to support four parallel terms, but an issue with the access to one of the memories prevented this. On the other hand, as is clear from the model, a 16-bank implementation does not result in operation close to I/O rates. For four parallel terms, this would require 64 banks; even for two parallel terms, the performance is 80% of the I/O rate. Our aim was not so much to achieve optimal performance as to implement and evaluate our novel design and compare it to the analytical model. We therefore decided to limit the number of banks to 16 to reduce the complexity of the design, as the implementation was undertaken as a summer project.

This means that there is a lot of scope for improving the current implementation.

(i) We will deploy our design on a PROCStar-IV board which does not have this issue, and thus we will be able to score 4 terms in parallel rather than 2 terms.

(ii) Even with a single SDRAM, we can be more efficient; the SDRAM I/O rate is 4 GB/s (according to the PROCStar-III databook); our current rate is only 1 GB/s. By demultiplexing the scoring circuit, it should be possible to increase this rate to 4 GB/s.

(iii) Combining both improvements, an improved design could score 16 terms in parallel. This will of course require a Bloom filter with more banks to reduce contention, but considering the current resource utilisation that is not a limitation. Consequently, the improved design should be able to operate up to 8× faster than the current design.

In terms of the analytical model itself, there is some scope for further refinement, in particular for the external access; we currently use a single access time for one and more hits. Just like for the Bloom filter, we can include a fixed cost for concurrent accesses on the external memory. We also want to refine the model to include the effect of grouping terms: that is, the $n$ parallel terms are usually grouped per two or four depending on the I/O width. This affects the waiting time on contention, as all terms in a single group need to wait before a new group can be fetched. Currently, the model

assumes all terms are independent. For the case of two terms, this assumption is corrects for more terms there is a slight underestimation of the access time in the case of contention. The counting problem for this case is complicated as it requires enumerating all the possible groupings and working out the effect if one or more accesses per group are in contention.

## 7. Conclusion

In this paper we have presented a novel design for a high-performance real-time information filtering application using a low-latency "trivial" Bloom filter. The main contribution of the paper is the derivation of an analytical model for the throughput of the application. This combinatorial model takes into account the access times to the Bloom filter and the external memory, the access probability, and the probability and cost of contention on the Bloom filter. The approach followed and the intermediate expressions are applicable to a large class of resource-sharing problems.

We have implemented our design on the GiDEL PROCStar-III board. The analysis of the system performance clearly demonstrates the potential of the design for delivering high-performance real-time search; we have shown that the system can in principle achieve the I/O-limited throughput of the design. Our current, suboptimal implementation works at 80% of its I/O rate, and this already results in speedups of up to a factor of 20 at 125 MHz compared to a CPU reference implementation on a 3.4 GHz Intel Core i7 processor. Our analysis indicates how the system should be dimensioned to achieve I/O-limited operation for different I/O widths and memory access times.

Our future work will focus on achieving higher I/O bandwidth by using both memory banks on the board and time-multiplexing the memory access. Our aim is to achieve an additional 8× speedup.

## Acknowledgments

The authors acknowledge the support from HP, who hosted the FPGA board and provided funding for a summer internship. In particular, we'd like to thank Mitch Wright for technical support and Partha Ranganathan for managing the project.

We'd like to acknowledge Anton Frolov who implemented the synthetic document model.

Wim Vanderbauwhede wants to thank Dr. Catherine Brys for fruitful discussions on probability theory and counting problems.

## References

[1] C. L. Belady, "In the data center, power and cooling costs more than the IT equipment it supports," *Electronics Cooling*, vol. 13, no. 1, 2007.

[2] W. Vanderbauwhede, L. Azzopardi, and M. Moadeli, "FPGA-accelerated information retrieval: High-efficiency document filtering," in *the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 417–422, September 2009.

[3] L. Azzopardi, W. Vanderbauwhede, and M. Moadeli, "Developing energy efficient filtering systems," in *the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '09)*, pp. 664–665, July 2009.

[4] V. Lavrenko and W. Bruce Croft, "Relevance-based language models," in *the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 120–127, September 2001.

[5] Lemur, "The Lemur toolkit for language modeling and information retrieval," 2005, http://www.lemurproject.org/.

[6] V. Kindratenko, R. Wilhelmson, R. Brunner, T. J. Martíez, and W. M. Hwu, "High-performance computing with accelerators," *Computing in Science and Engineering*, vol. 12, no. 4, Article ID 5492949, pp. 12–16, 2010.

[7] GiDEL Ltd, "PROCStar III, Data Book," September 2009.

[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[9] Altera Corp, "Stratix III, Device Handbook," July 2010.

[10] G. Andrews and K. Eriksson, *Integer Partitions*, Cambridge University Press, 2004.

[11] C. Chen and K. Koh, *Principles and Techniques in Combinatorics*, World Scientific, 1992.

[12] R. M. Losee, "Term dependence: a basis for Luhn and Zipf models," *Journal of the American Society for Information Science and Technology*, vol. 52, no. 12, pp. 1019–1025, 2001.

[13] M. A. Montemurro, "Beyond the Zipf-Mandelbrot law in quantitative linguistics," *Physica A*, vol. 300, no. 3-4, pp. 567–578, 2001.

[14] W. Press, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 2007.