Hindawi Publishing Corporation
VLSI Design
Volume 2014, Article ID 712085, 11 pages
http://dx.doi.org/10.1155/2014/712085

Research Article

A Low-Power Scalable Stream Compute Accelerator for General Matrix Multiply (GEMM)

Antony Savich and Shawki Areibi

School of Engineering, University of Guelph, Guelph, ON, Canada N1G 2W1

Correspondence should be addressed to Antony Savich; asavich@uoguelph.ca

Received 6 August 2013; Revised 15 December 2013; Accepted 18 December 2013; Published 24 February 2014

Academic Editor: Jim Ching-Rong Lin

Copyright © 2014 A. Savich and S. Areibi. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Many applications ranging from machine learning, image processing, and machine vision to optimization utilize matrix multiplication as a fundamental block. Matrix operations play an important role in determining the performance of such applications. This paper proposes a novel efficient, highly scalable hardware accelerator that is of equivalent performance to a 2 GHz quad core PC but can be used in low-power applications targeting embedded systems requiring high performance computation. Power, performance, and resource consumption are demonstrated on a fully-functional prototype. The proposed hardware accelerator is 36× more energy efficient per unit of computation compared to state-of-the-art Xeon processor of equal vintage and is 14× more efficient as a stand-alone platform with equivalent performance. An important comparison between simulated system estimates and real system performance is carried out.

1. Introduction

Matrix manipulation operations are crucial steps in many types of applications ranging from machine learning techniques such as artificial neural networks to image and signal processing. One of the most fundamental actions within these algorithms is matrix multiplication. The complexity of matrix multiplication is $O(N^3)$, where N is the dimension of a square matrix. Accordingly, it requires substantial computing power especially when the matrices are quite large such as in medical imaging, 3-D image manipulation, or even in complex optimization problems that require solving a set of linear of equations. This paper proposes an efficient hardware accelerator that differs from previous research in the following areas. (i) The proposed architecture is general enough so as to be efficiently utilized in any application incorporating matrix-matrix or matrix-vector multiplication. (ii) The proposed architecture is highly scalable; it is capable of operating on smallest to largest devices, single or multiple FPGAs. (iii) Not only the performance but also the power and energy tradeoffs in using the proposed design are studied. (iv) The systems consume considerably less power than conventional general purpose processors which entails it to be used as an embedded system.

To the best of our knowledge this paper is the first to address performance, scalability, and power at the same time. The remainder of this paper is organized as follows. In Section 2 essential background on matrix multiplication and necessary literature review of current work are addressed. The main architecture proposed in this paper is presented in Section 3. Experimental results and analysis are then presented in Section 4. Finally, the paper is concluded in Section 5 and directions of future work are discussed.

2. Previous Work

Traditional Von Neumann architectures suffer from a severe bottleneck seriously limiting the effective processing speed when the CPU is required to perform number crunching on large amounts of data. This can be attributed to the sharing of the bus between the program memory and data memory. Improving the performance of the computer system can be achieved by exploiting the parallelism in the form of spatial and temporal techniques. Temporal parallelism tends to use

multistage pipelining to partition the application into several phases that can run simultaneously. Spatial parallelism on the other hand tends to use multiple cores, duplicated functional units, and multiprocessors to achieve speedup. The right balance between data flow and computational resources is essential in highly parallel systems.

In [1], new algorithms and architectures for matrix multiplication on configurable hardware were developed. The proposed designs significantly reduce the latency as well as the area. Benchmarks used were limited to latency on small matrix sizes and designs, and only simulated results were reported using Xilinx XC2V1500, without considering implementation constraints in detail. In [2] a preliminary design and FPGA implementation of dense matrix-vector multiplication for use in an image processing application are presented. Results obtained demonstrate that it can provide a maximum throughput of 16970 frames per second using very small dimensional data. Neither comparison with state-of-the-art processor is carried out, nor a real FPGA system is considered. In [3] the authors propose two parallel matrix multiplier implementations. The first proposed multiplier is dedicated to 3D affine transformation, while the second one is for colour space conversion. The two multipliers are implemented using Xilinx CoreGen and Celoxica fixed-point library. The architecture is mapped onto a Xilinx XCV2000E Virtex FPGA. The authors compare the performance of the dedicated matrix multiplier with an ATI RADEON FSC 32 MB graphics card. Results obtained in terms of minimum period and computation time indicate that the GPU is approximately twice as fast. No power consumption of the system has been published.

Authors in [4] discuss a solution to the first MEMOCODE hardware/software codesign contest which posed the problem of optimizing matrix-matrix multiplication such that it is split between the FPGA and PowerPC on a Xilinx Virtex II Pro. The given test framework featured a software only implementation and their work was to use the FPGA as an accelerator to decrease execution time. The operation was performed using fixed size square matrix blocks, and the architecture is not suitable to scale to different problem sizes of future architectures. It is also difficult to duplicate the work since insufficient details are provided in the paper. In [5] authors propose a high performance least squares solver on FPGAs using the Cholesky decomposition method which achieves speedup compared to a software implementation on a Pentium 4. However, benchmarks used were small (128×128) and their proposed architecture is neither scalable nor flexible and is limited to solve least square problems.

In [6] a reconfigurable device which significantly improves the execution time of the most computationally intensive function of three of the most widely used face recognition algorithms is presented. At the heart of these algorithms is multiplication of very large dense matrices. Authors claim that their system achieves a 550× speedup over PC. However, the PC implementation used in comparison is orders of magnitude slower than can be obtained with Matlab on similar vintage hardware, as we will demonstrate. Their design has very coarse grain scalability, is not efficient in resource use, thus not suited for embedded system implementations. No demonstration in use for face recognition algorithms is presented, because the functionality is only simulated. The work in [7] proposes a matrix multiplication architecture using systolic arrays on an FPGA which increases the computing speed by combining the concept of parallel processing and pipelining into a single design. Results obtained were merely based on simulations and no comparison or speedup was reported with respect to state-of-the-art general purpose processor. Their design uses a high data bandwidth to computation ratio, which hinders estimated scalability of this design. [8] introduces a scalable macropipelined architecture to perform floating point matrix multiplication, which aims to exploit temporal parallelism and architectural scalability. The architecture is capable of achieving 12.18 GFLOPS with 32 processing elements (PEs) or about 1.90 GFLOPS per PE per GHz performance. However, results for large matrix multiplication were not implemented on the FPGA but rather simulated due to resource limitation on the ML507 development board used.

In [9] the design and implementation of a sparse matrix-matrix multiplication architecture on FPGAs is presented. It is one of the few publications that studies performance of the design, in terms of computational latency, as well as the associated power-delay and energy-delay tradeoffs. Comparison neither in terms of performance with general purpose processors nor in terms of speedup achieved is presented. Authors in [10] present two designs for matrix multiplication, optimized for implementation on high-end FPGAs. The proposed architecture with 40 PEs and a design speed of 373 MHz achieved a performance of 29.8 GFLOPS. Results published were based on simulations only using Xilinx ISE 10.1 and ModelSim 6.2e. No full system design constraints were taken into consideration when developing or simulating the performance of the two systems.

Two recurrent themes emerge from the review. The first theme is based on assumptions (simulation) versus real system considerations (actual implementation). We demonstrate that the attachment and overall system design play a significant role in achieving targeted performance and scalability. Yet few works consider it in developing new proposed architectures, and fewer still utilize functional prototypes in claims of improvement and comparisons. The second theme is power consumption. There is little emphasis given to this topic in the literature; yet it is very important for systems targeting embedded reconfigurable applications.

3. Proposed Architecture

Fine grain scalability is one of the main themes of our work. In order to accomplish this goal, an architecture needs to be adaptable to the resources available in the target platform, in this case, an FPGA. A variable number of processing elements are required to manipulate data, and this number should optimally change in unit increments to achieve best resource utilization and performance efficiency.

At the same time, the I/O interface is the typical bottleneck of most if not all high performance GEMM as well as other compute accelerators. The external I/O interface is highly important to consider, and, in the case of this work,

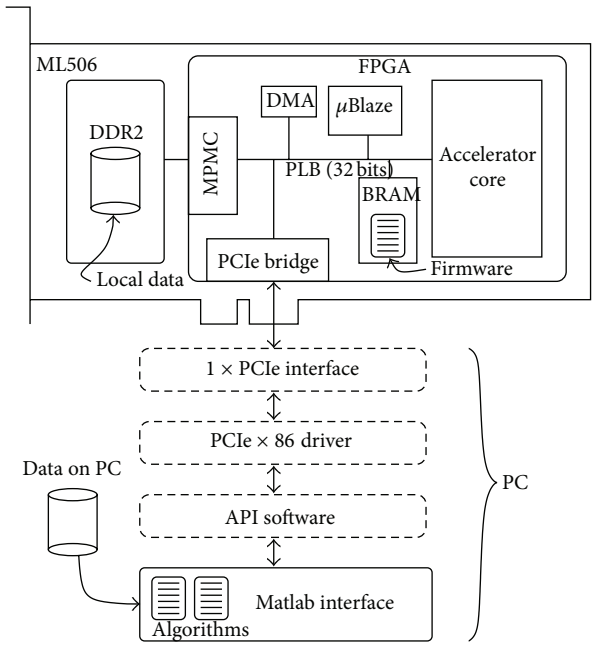


Figure 1: System architecture.

the architecture is built from the attachment system and interface inwards. A stream based dataflow architecture is chosen. The I/O ports for a single core, composed of multiple processing elements, are represented with few unidirectional data streams. The processing elements inside the core are systolically attached, allowing the fine architecture granularity and high data reuse within an application. This approach performs well in many applications, including GEMM.

Using this simple unified general purpose I/O approach, system scalability is further exploited by matching attachment system's I/O capabilities to the algorithm's computational granularity to achieve enhanced performance using minimal resources. In systems with limited data I/O, a single small or large core can compute larger data sets efficiently. On the other hand, in systems capable of sustaining multiple I/O stream attachments, multiple heterogeneous cores can perform a wide range of operations, from large to tiny, with high utilization efficiency. Multichip multicore system configurations are achievable for best performance and flexibility.

The core is also scalable in the representation of data, where operand arithmetic representation is a fully parameterized value. The data can be represented using fixed or floating point formats, which changes the type of underlying hardware blocks used. The bit widths of the representation, fixed or floating point, can also be modified at synthesis time and can be different for each argument and intermediate value in computation.

In the next few sections the architecture of both the compute core and the fully functional prototype system will be explained in detail.

3.1. Overall System Architecture. The overall system architecture is demonstrated by Figure 1. Matlab 2010b is used as the interface to the accelerator. Matlab employs vendor specific BLAS libraries to attain top GEMM speed for measuring CPU performance and provides a uniform verification and benchmarking platform for the FPGA board. API software is developed to interface between Matlab and the PCIe driver for the accelerator. It is used to transfer data and control operations over a 1× PCIe link to the Xilinx ML506 board (more details in Section 4). The accelerator platform can perform computations on any data size using any compute core configuration. It is limited only by the size of the DDR2 SODIMM selected. The board is currently fitted with 256 Mb DDR2-400 RAM.

Table 1: Naming convention.

	Hardware port	Corresponding operand
C	Cache prefetch stream	Y
S	Compute stream	X
B	Auxiliary stream	α, β, W, Z'
P	Result stream	Z, Z'
i	Designates input	
o	Designates output	

The accelerator can also perform computation on data in PC's memory directly, being limited by the size of PC memory hierarchy. This approach is not used since accelerator runtime I/O requirements exceed the bandwidth of the 1× PCIe interface provided by the ML506.

It must be noted that the choice of a PCIe attachment interface and a Matlab API is based solely on the simplicity of experimental setup. The same data sets can be used in generating both PC performance and FPGA results. The results of both computations can be easily compared using the same tool chain for validation and numerical accuracy. The real target for the FPGA architecture proposed in this work is low-power embedded applications requiring high performance matrix computation acceleration. Other choices of attachment interfaces range from GigE/10GE for remote data storage to direct connection with live sources of data, such as cameras and sensors. The FPGA system implemented in this work to demonstrate the proposed architecture is independent of the PC itself and can be used entirely without it. The on-board MCU and software are self-sufficient and allow for completely independent operation regardless of the source of data used.

3.2. Core Architecture. The internal architecture is presented bottom-up in this section. Table 1 lists the notation used in the figures that follow. $Z = \alpha XY + \beta W$ is used as an illustration, where α and β are scalars and all other operands are matrices. A $(')$ represents a partial result; for example, Z' is the partial product of an incomplete block operation $X'Y'$.

The accelerator adopts a dataflow architecture. Most control functions are driven by the data flow itself. Computations on matrices are performed in blocks, and the blocks are further subdivided into sequences of rows or columns. The inputs are pushed into several input chains connected via the stream input ports. The data is presented to each

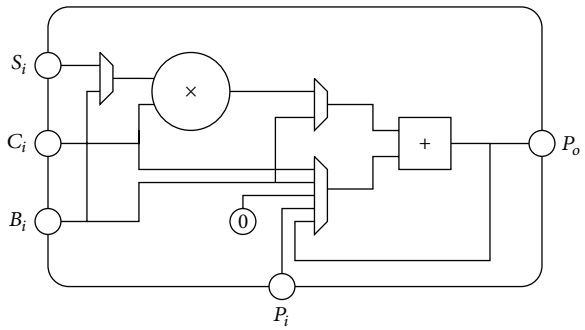


Figure 2: The miniprocessing element (mPE).

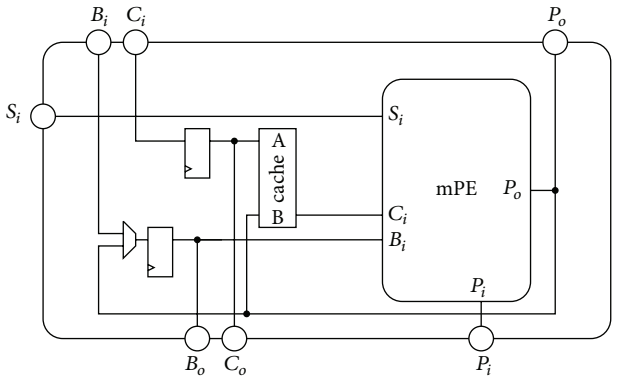


Figure 3: Processing element (PE).

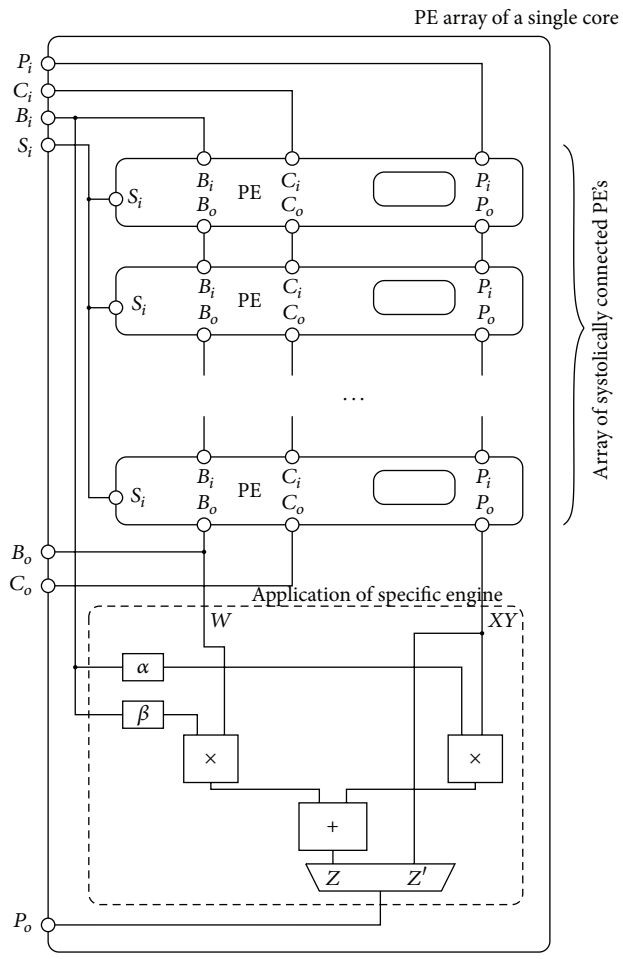


Figure 4: PE systolic array with ASE.

processing element in the chain in succession. The processing elements use the data values flashed on their inputs in the chain together with values stored in their local caches to compute required operations. When results are formed, they are either stored back into local caches for further processing or cached onto the output chains, which are then streamed sequentially out of the accelerator. The seemingly sequential nature of I/O interaction with the core significantly simplifies and relieves bandwidth requirements to get the data to and from the accelerator, while at the same time allows for highly scalable parallelization of computation on many data elements simultaneously by arrays of simple processors.

At the lowest level of hierarchy, the structure of a miniprocessing element (mPE) is demonstrated in Figure 2. It contains all basic hardware necessary to perform GEMM effectively. The extra pathways, such as a *Pi-Po* stream, give an I/O efficiency improvement by enabling in-stream selection of $Z = XY$ or $Z = X^TY$ operators. Operand row-order in memory is maintained, and an extra row-column reorganization step is eliminated by in-stream selection of compute pathways (the same PE accumulation or PE-PE transfer accumulation).

The mPE is combined with cache and stream interfaces to form the processing element (PE), as shown in Figure 3.

The auxiliary stream (B) is used for loading partial products, performing auxiliary scalar or element-wise operations, or offloading full or partial results from the accelerator. The preload and offload operations on this stream can be performed simultaneously, reducing the number of I/O ports, stream registers, and total I/O cycles.

The PEs are arranged in a systolic array to form the basis for the compute core, as demonstrated in Figure 4. A single core contains one stream attachment interface (to any standard bus), a control block and queue, and a systolic PE array. In the current prototype, a PLB attachment is used (Figure 5).

A GEMM $Z = \alpha XY + \beta W$ operation would involve the steps in Algorithm 1.

The degree of parallelization among the operating steps is variable and depends on the size of the PE array available in the core. The time complexity of the matrix multiplication is $O(nmp/\text{PE})$, where the matrix sizes of the operands are $n \times m$ and $m \times p$, and PE is the number of processing elements available in the core.

3.3. Application Specific Engine (ASE). Notice that the PE array has an application specific engine (ASE) attached in-stream, as seen in Figure 4. The ASE is configurable to perform any scalar, element wise, or benchmarking operations directly in hardware concurrently to a GEMM operation in progress. This is a novel ability, which packs many

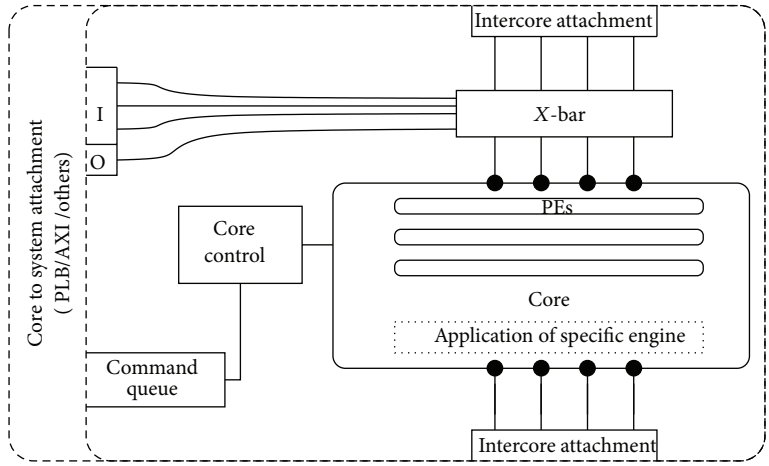


Figure 5: Single core.

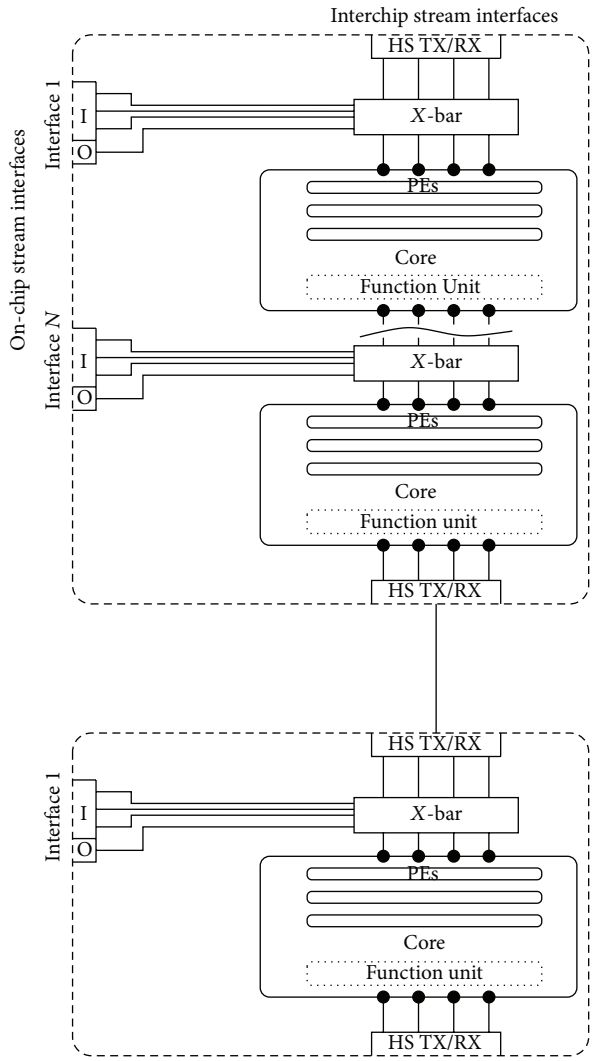


Figure 6: Multicore/multichip architecture.

noncompute but I/O intensive operations concurrently with high I/O and high computation core functions in-stream. By analyzing target applications, sets of GEMM and any number of related auxiliary operations can easily be extracted and grouped. The basic configuration supporting the $Z = \alpha XY + \beta W$ operation is demonstrated in Figure 4, but this is only the simple case. The ASE allows performing any group (GEMM + AUX) as a single operation in the core, reducing total computation time and I/O bandwidth requirements significantly. In many algorithms, all arithmetic operations can in fact be decomposed into such groups. Computational performance in such cases is equal to that of the underlying matrix operations, with auxiliary computation and bandwidth overhead effectively hidden.

3.4. Multicore/Multichip. The same attachment interface, as is illustrated in Figure 5, can be used at run-time to connect multiple cores together, forming a chain or star topology of systolic arrays, as demonstrated in Figure 6. There is no advantage in using multiple stream interfaces per core, since a single stream interface per core is sufficient for full utilization. However, in systems where available I/O bandwidth is high, and multiple stream ports can be used, there is an advantage in implementing multiple cores. A heterogeneous PE configuration enhances computational efficiency where small and large matrices can be computed in parallel, each on a core of most efficient size for the underlying data set. Multiple small matrix/vector operations can be performed concurrently on multiple small size cores, increasing parallelism scalability. For large data operations at various algorithm steps, multiple cores can be connected together on the fly, whether such cores reside on one or multiple FPGAs, to facilitate high data reuse and boost available parallelism and hence performance on such large sets.

3.5. Sparse Matrix Manipulation Extensions. This work demonstrates a functional dense matrix compute architecture

```
divide X and Y into blocks X′ and Y′ of size (PE × cache depth);
For each block of X′ and Y′ do
    prefetch Y′ into cache via stream C;
    preload any W, α, β or Z′ to stream B;
    for each row of X′ do
        stream new elements of the row of X′ via S;
        multiply-accumulate elements of X′ and Y′ across PEs;
        if Z′ contains final elements of XY then
            shift new partial results of Z′ from PEs via P;
            perform scalar operations using α, β and
            W at output of P and B via ASE (§ 3.3);
        else
            shift new elements of Z′ from PEs via P
            to memory or cache;
        end
    end
end
```

Algorithm 1

and prototype. Sparse matrix computation also plays a significant role in scientific applications. Sparse matrix computation performance is highly dependent on the compatibility of the compute architecture with the data representation format. While outside the scope of this article, the stream architecture presented here is well suited for acceleration of sparse matrix computation with in-stream sparse format processing modules. Future work is planned on the proposed system to move from a dense only to sparse/dense matrix computation capability within a unified architecture. This is of high significance as many algorithms typically require both dense and sparse matrix operators that can benefit from acceleration. The typical case is to implement two separate custom cores or processors to perform each type of operation separately, with only half resources per each core achieving half of potential performance. Using a unified compute core will significantly improve resource utilization and flexibility, as well as boost overall performance.

4. Results

A comparison between the proposed FPGA system and a full featured PC is carried out in this section. The FPGA system is mapped to the Xilinx ML506 board with XC5VSX50 FPGA manufactured using 60 nm silicon process and having 288 DSP blocks, 264 18 Kb Block RAMs, 8,160 slices, 256 Mb external DDR2-400 RAM, and 1× PCIe port. The PC system used is a Dell T7500 workstation, with 2 Gb DDR3-1333 RAM and a quad core Intel Xeon E5405 2 Ghz processor with 2 × 6 Mb L2 cache manufactured using 45 nm silicon process.

Our current results are based on an FPGA system that is configured with a single 204 PE core, a PLB attachment, a single 32 bits Xilinx PLB DMA controller, a MicroBlaze MCU, 1 lane PCIe to PLB bridge, and one PLB to DDR2 memory controller (MPMC) port. The core is configured to use 16 bits wide fixed point 1-3-12 arithmetic representation, giving $[-2^3, 2^3)$ range and 2^{-12} uniform precision across all hardware channels for simplicity. A full precision accumulator, in this case 32 bits wide, is used to eliminate accumulation errors (only data conversion and result truncation errors are present). This representation is adequate for many algorithms. Larger representations, up to 18 bits wide for cache stored multiplicands and 25 bits for streamed multiplicands and partial products, with appropriate lossless accumulators, are feasible with a negligible increase in resource consumption on the same FPGA. The Virtex 5 FPGA contains 18×25 hard DSP blocks and 18 bits wide Block RAMs in the correct BRAM-DSP proportions, so that one BRAM-DSP pair is used per PE. Wider fixed point and floating point representations require multiple BRAM-DSP pairs per PE. An investigation of the effects of larger representations on resource consumption and performance reduction due to lower PE count will be considered for future work.

In all comparisons, the PC's task load outside of the computation at hand was minimized by halting all nonessential services and processes, to eliminate distortion of results from any nonrelated computation. The idle workload, before and after tests, was negligible at below 1%.

4.1. Power Consumption. Power is at the heart of this comparison. Hence, we present our findings and demonstrate that the FPGA embedded platform significantly outperforms the PC. Table 2 summarizes the results. Not only the up-front power dissipation is greatly reduced by performing computation on FPGA at similar performance levels but the form factor is also a significant advantage. Taking advantage of the flexibility and scalability of the accelerator architecture presented here, large scale high performance implementations can be implemented at the embedded scale, independent of the PC.

The power measurement for the PC are obtained by measuring the total power input at the mains ($P = VA$) and for FPGA at the board power input. Unfortunately there is no facility to measure individual FPGA power rail consumption on the ML506. FPGA power reported is a board level measurement. Even though the FPGA board is

Table 2: Power consumption, FPGA versus PC.

	FPGA	PC	Δ
Idle when configuring	5.70 W	117.4 W	21×
Idle (configured)	10.26 W	117.4 W	11×
uBlaze running, core idle	10.40 W	117.4 W	11×
Computation (system total)	11.07 W	164.8 W	14×
Overhead cost	10.4 W	117.4 W	11×
Computation cost	0.66 W	47.4 W	72×
Energy adjusted (J/GMAC)	0.101	3.62	36×

connected to the PC via the PCIe bus, it draws no power from the PC via that interface. The only associated power connection with the PCIe interface is in the signal drivers, with power attribution negligible.

The PC, when idle, consumes 117.4 W of power (with no FPGA board). In comparison, the FPGA board consumes 5.7 W when being configured (peripheral device clock nets are down), 10.26 W when configured and MCU not running, and 10.4 W when MCU is running awaiting operations. The FPGA board consumes 11.07 W in full computation, at approximately 1 : 5 system to clock ratio (core clock is gated) to achieve similar performance with PC consuming 164.8 W. When comparing energy consumed per unit of computation (J/GMAC), the FPGA is 36× more energy efficient versus PC, not including overhead. This result demonstrates the suitability of the proposed architecture in embedded systems, giving no sacrifice to a wide application range while maintaining excellent performance.

4.2. Performance. A number of datasets are selected to report system performance and compare it to the performance of the PC. Results are listed in Table 3. It should be noted that the performance currently achieved by the FPGA system is nearly that of the PC. Yet the emphasis of this work is to produce a low-power, highly scalable and high performance compute core targeting small form factor embedded systems. System attachment of the core, even at these levels of performance, is still far from optimal. A straight 10× improvement can be achieved by replacing the stock Xilinx 32 bits DMA controller with an efficient alternative.

Performance is measured in Giga Multiply ACcumulates per Second (GMACS). 1 GMACS is equivalent to 2 GFLOPS when using floating point arithmetic and 2 GIOPS (Giga Integer Operations Per Second) when using fixed point arithmetic. The best and worst performance are highlighted in bold. The core clock is gated to compensate for the inefficiencies of the Xilinx supplied DMA mechanism. In the current best case, the core is clocked once (1 word of data available) every 5 system cycles. The worst ratio on this dataset is 7.

The performance listed in Table 3 is achieved with a single 204 PE core using an inefficient DMA controller for streaming the data. All system components have been optimized to operate at 200 MHz. A basic DMA block is used to pump data from on-chip memory controller to the stream core via the 200 MHz 32 bits PLB bus. The DMA controller used is the standard PLB DMA IP core provided by Xilinx in EDK 12.4 suite. It can transfer only 32 bit words of data per bus cycle; thus wider bus configurations are not effective. It is a half duplex core supporting 16 word burst transfers and in testing achieves only 1 word transferred per 5 bus cycles; this explains the 1 : 5 system to core ratio. This roughly translates to a bandwidth of approximately 200 Mb/s or 50 Mwords/s. The memory controller and on-board DDR2 is capable of 3.2 Gb/s bandwidth. Core frequency and vendor dependent optimizations were not the emphasis of this work, as the limiting factor is this attachment system. Nevertheless, the core itself without an attachment interface can operate at 345 MHz (synthesis estimate) on the currently used FPGA without any optimization. Based on the simplicity of design, the fact that all critical path components reside in the mPE (which has been designed to fit completely within hard DSP slices), there is no doubt that a vendor specific optimization achieves the advertised 550 MHz barrier. XST synthesizer from Xilinx is not able to interpret any vendor independent VHDL code of reasonable functional complexity cleanly into DSP slices at this time.

At this frequency and assuming an improved DMA engine, the core can achieve performance between 19.5 and 89.1 GMACS (equivalent to 39–178 GIOPS) on the current board. This chip's absolute maximum estimated performance, given no attachment overheads and other system constraints (thus increasing PE count to 288 and using 4 independent stream engines) in a "perfect" simulated system is 158 GMACS or 316 GIOPS—a 12× improvement over the 4 core PC.

Given the current core implementation and ignoring an inefficient DMA engine and vendor independent VHDL to hardware mapping, Table 3 also lists the acceleration of the core of the PC using a 0.5 system to core ratio (400 MHz) assuming an upgraded DMA controller. The result of simply substituting an appropriate DMA controller versus stock Xilinx offering is an acceleration of 4.3–6.8× over the PC at a core clock frequency five times slower than that of the PC. The theoretical performance improvement of FPGA versus this PC can be reestimated at 30–40× when considering the largest Virtex5 device (sx240t). It would be a fair comparison, as the Xeon processor used here is the largest in the series of this vintage and is manufactured using 45 nm technology versus 65 nm for the FPGA. However, the emphasis of this paper is on real results of performance and most notably power efficiency in equivalent vintage and cost systems, and not on estimates.

4.3. Resources and Performance Density. Resource consumption is directly proportional to performance gain. In designs where PEs consume more resources, a smaller number can be placed using the same footprint, thus reducing performance density. Because this work is proven using a fully functional prototype, and not simply simulations and estimates, effects of the attachment system components need to be considered. The design choice of a particular attachment system can make a difference in the final device performance, as is illustrated here. Table 4 shows resource consumption by PE, core, and system, based on the current mapping using a maximum

Table 3: FPGA accelerator performance versus high-end quad core PC.

Matrix size	Ops (MMACs)	PC time (s)	FPGA compute time (s)	FPGA system cycles	FPGA core cycles	System to core ratio	PC perf (GMACS)	FPGA Perf (GMACS)	PC versus FPGA	FPGA perf at 1 : 2 rat.	FPGA versus PC at 1 : 2 rat.
4096 × 4096	68,700	5.25	12.7	2.53 G	491 M	5.15	**13.1**	5.43	2.4×	56	**4.3**×
2048 × 2048	8,500	0.704	1.65	328 M	63.9 M	5.13	12.1	5.22	2.3×	54	4.5×
1024 × 1024	1,070	0.120	0.221	44 M	8.62 M	5.10	8.91	4.86	1.8×	50	5.7×
512 × 512	134	0.0235	0.0383	7.6 M	1.37 M	5.54	5.70	3.50	1.62×	39	**6.8**×
128 × 128	2.1	0.00081	0.00207	415 k	59 k	7.03	**2.59**	**1.01**	**2.56**×	**14**	5.5×
4080 × 1024	17,045	1.403	2.629	526 M	105 M	5.00	12.1	**6.48**	1.87×	**65**	5.3×
204 × 1024	42.6	0.0076	0.0121	460 k	2.4 M	5.21	5.58	3.5	**1.59**×	36	6.6×

The bold font emphasizes important entries for comparison reasons.

Table 4: Resource utilization.

	PE	Core	System	Util'n	Available on XC5VSX50
LUTs	51	10,472	23,767	73%	32,640
FFs	34	7,079	19,947	61%	32,640
Slices	17	3,497	7,910	97%	8,160
BRAM	0.5	102	132	100%	132
DSP	1	204	207	79%	288

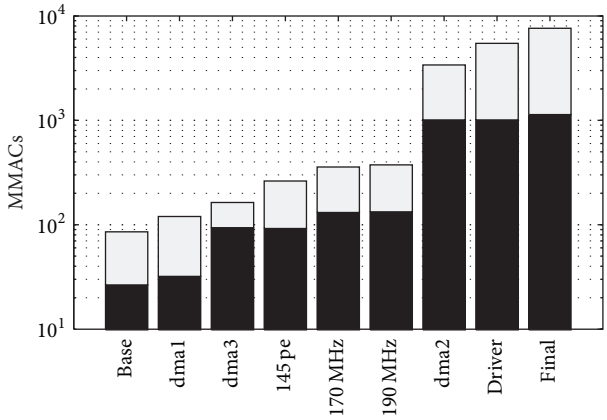


Figure 7: Optimization effect on performance.

number of PEs for a PCIe based system on a Xilinx ML506 board.

When the synthesizer, as is typically reported in simulation-based works, uses a certain number of LUTs and FFs below the device capacity, a design can still easily fail placement on a real system. Routing constraints, contrasting control sets, and timing expectations must be carefully considered to make sure that a design is actually feasible. In Table 4, this point is clearly demonstrated with the core using only 32% of LUTs and 22% of FFs, but more than 42% of available slices (4 LUT-FF pairs on Virtex 5 devices) because some LUTs and FFs cannot be paired. Further, the final system uses 97% of all available slices on the chip. In this design, the remainder of slices is occupied by the MicroBlaze MCU (~5%), DDR2 Memory Controller or MPMC (~15%), and PCIe-PLB Bridge (~30%). Approximately 30 36 Kbit BRAMs are used by the system in the MicroBlaze and operating memory. Operating memory that is used to store the core and system drivers and firmware can be placed in off-chip SRAM. This, however, does not free up a significant amount of resources in comparison, as an additional SRAM controller will occupy scarce slices and will reduce the maximum number of PEs placeable.

The system performance is directly proportional to the available resources on the chip. The hardware cost complexity consists of system overhead and core components. System overhead is approximately 4.5 k slices, 30 BRAMs, and 3 DSP. The overhead configurable resources (slices) are taken up primarily by the multiport DDR memory controller and the PCIe interface hardware. The BRAMs are used for the on-chip memory, which contains the required software to drive the accelerator board and communicate with the PC. Three DSPs are used in the MCU. The core consumes approximately 17 slices, 1 DSP, and 0.5 BRAM per PE. Given the 6.5 GMACS actual (at 5 : 1 system to core clock ratio) and 65 GMACS DMA upgraded (at 1 : 2 system to core clock ratio) performance for the 204 PE system, this results in the consumption of 530 (53) slices, 31 (3) DSP, and 16 (1.6) BRAMs per each 1 GMACS of actual (upgraded) performance required. The consumption is nearly linear with the computation of matrices of any size larger than the number of PEs in the system. This result is obvious as it becomes inefficient to use a long systolic array for computation of small matrices. Special data paths can be added to masquerade a long systolic array as a short one to improve efficiency, but a better solution is to use a heterogeneous multicore architecture where a variety of array sizes can simultaneously provide improved performance in two dimensions: by improving efficiency of smaller matrix computations in each core and at the same time performing multiple small matrix operations in parallel. The smaller cores can then be linked together into a large array on the fly to compute large matrix operations with maximum performance.

4.4. Design Exploration. It is important to highlight the incremental optimization steps taken to produce the highest performance from the same hardware available. Figure 7 shows the effect of revisions on performance, for both minimum and maximum obtained on the data set presented in Table 3.

The base accelerator design is 80 PE, 125 MHz polled MCU I/O system. The final system is 204 PE, 200 MHz, with DMA dataflow. Two key factors play a crucial role in extracting maximum performance: (i) attachment interface and (ii) PE placement optimization. Three stages of dataflow—cache prefetch (dma1), stream compute (dma2), and result offload (dma3)—are converted in sequence from polled to burst DMA transfers over on-system PLB bus. While dma1 and dma3 are relatively easy to implement since they require only small additional control hardware to operate the core in-transfer, dma2 requires more careful design and tuning. Converting stream compute dataflow to automatic operation, without MCU control, achieves the most performance improvement as a single optimization step. Bandwidth requirements for compute during DMA stage 2 are similar to cache prefetch. Control bandwidth reduction is attained in dma2 by automating more complicated control in hardware. This reduction is of the same order as the data bandwidth required for the compute operation. It also eliminates MCU cycles required for command computation in driver software—a comparatively slow operation.

Several iterations of frequency improvements provide a marginal effect. Several data reuse enhancements in the block matrix multiplication operation are implemented in the driver. Performance is enhanced when vendor independent code is optimized further in order to enable more efficient mPE to DSP slice mapping by the synthesizer. An initial 80 PE system is boosted to 204 PEs hitting the resource wall due to auxiliary (MCU/PCIe/MPMC) slice and BRAM utilization. Virtex 6 and 7 devices provide a greater BRAM to DSP and

slice to DSP ratios. In these devices, the number of DSP blocks determines placeable PE number, and it is in general significantly larger than in Virtex 5 devices.

4.5. Actual versus Estimated Performance. Many works, published in the literature, present results based on simulations; that is, actual implementation is neither verified nor demonstrated, and no end-to-end system constraints are considered. This work, however, provides experimental foundation of why not all typical assumptions used for extrapolated performance hold true.

For example, in [6], a 550× speedup is claimed using a block matrix algorithm for matrix multiplication and necessarily square architecture construction, with very high granularity of scalability. Only small square blocks of [16, 16], [32, 32] or [64, 64] elements can be executed at a time. No prefetched data reuse is exploited. The architecture, when scaling to larger chips, is unable to take full advantage of existing resources, with the cited simulated design occupying 73% of any one resource at most. In an attempt to repeat presented results, when PC performance is compared with the FPGA system and a 550× improvement is claimed, we provide an argument that not all estimates can be taken at face value. We obtained 100–200× better results on our PC versus PC results claimed in the 550× improvement comparison, based on their dataset sizes. Our PC is demonstrably of equivalent vintage to the FPGA board we are using. The FPGA device selected in simulations of [6] is 5× larger than our device, and no attachment hardware resources are reported. Therefore, for a fair comparison, the architecture in [6] is generally slower than equivalent vintage CPU based system. Comparing further the reported performance in [6] on medium size (1024 × 1024) matrices and using equivalent assumptions about an optimal attachment system, our novel architecture is about 5–10× faster. Our design would achieve 484 GMACS or 968 GIOPS on the XC5V240T FPGA, versus 37GMACS or 74 GIOPS as presented in [6].

In [4], 396 MMACS of performance on large matrices is reported. Our design is about 10× faster using wall clock on current hardware than [4]; our PC results are about 1,000× faster than reported in [4] using similar vintage hardware, a significant divergence. In contrast, using estimated performance as above, our design would be more than 1,000× faster than that claimed in [4]. At the same time [6] report being 100× faster than [4], while claiming a 550× improvement over state-of-the-art PC performance. Similarly, [7] report 2.95 GMACS of estimated performance for one design and 14.9 GMACS for a second design using an assumed FPGA system with a large Vertex 5 chip, the same as used in estimates of [6], and zero attachment and communication system overhead. While this result appears 2× faster than our fully functional prototype, implemented on an FPGA having five times fewer resources and half the clock rate, in comparison to our extrapolated performance on equal hardware with similar system assumptions, it in fact is about 30× slower.

Estimating power consumption of a complex simulated system on FPGA is very difficult and often not very accurate. Power estimates are further complicated when moving to the board level to provide full system power cost, which is essential for all embedded applications. With a physical implementation, we are able to demonstrate true device performance and system power analysis. This together has not been done to date in the literature, due in large part to the simulated nature of the designs.

5. Conclusions

In this paper, a novel scalable and low-power stream compute accelerator has been presented targeting algorithms based on GEMM operations. The current fully-functional stand-alone prototype, using a mid-range FPGA, is demonstrated to be at par with the performance of a high-end quad core CPU platform of equal vintage. There is an even higher estimated potential for performance, to be investigated in future work, when application specific auxiliary computations are performed in-stream with matrix multiplication—an area where highly optimized CPU loses its advantage considerably.

The proposed novel architecture demonstrates excellent scalability and can be easily ported into heterogeneous and multichip high performance architecture domains for embedded computing. The most important benefit demonstrated here is the system's ability to deliver excellent performance at a fraction of the power and energy cost of a similar general purpose system, while offering a path to maintain generality of accelerated computation for a wide range of embedded applications. The developed accelerator system is 72× more power efficient at current levels of performance in computation when ignoring overhead, 36× more energy efficient per unit of computation, and 14× more efficient in full system power consumption in comparison of a PC versus our configurable FPGA platform.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] J. Jang, S. Choi, and V. Prasanna, "Area and time efficient implementations of matrix multiplication on FPGAs," in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT '02)*, pp. 93–100, Hong Kong, China, December 2002.

[2] A. Qasim, A. Telba, and A. AlMazroo, "FPGA design and implementation of dense matrix-vector multiplication for image processing application," in *Proceedings of the World Congress of Engineering and Computer Science (WCECS '10)*, vol. 1, pp. 1–4, San Francisco, Calif, USA, October 2010.

[3] F. Bensaali, A. Amira, and A. Bouridane, "Accelerating matrix product on reconfigurable hardware for image processing applications," *IEE Proceedings—Circuits Devices and Systems*, vol. 152, no. 3, pp. 236–246, 2005.

[4] N. Dave, K. Fleming, M. King, M. Pellauer, and M. Vijayaraghavan, "Hardware acceleration of matrix multiplication on a xilinx FPGA," in *Proceedings of the 5th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE '07)*, pp. 97–100, Nice, France, June 2007.

[5] D. Yang, G. D. Peterson, H. Li, and J. Sun, "An FPGA implementation for solving least square problem," in *Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 303–306, Napa, Calif, USA, April 2009.

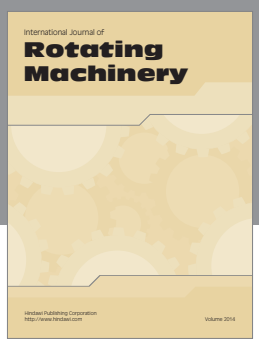
[6] I. Sotiropoulos and I. Papaefstathiou, "A fast parallel matrix multiplication reconfigurable unit utilized in face recognitions systems," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 276–281, Lausanne, Switzerland, September 2009.

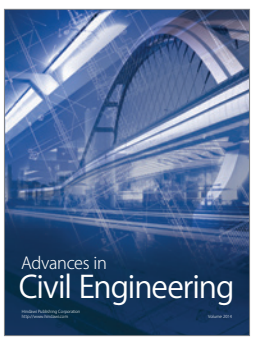
[7] M. Vucha and A. Rajawat, "Design and FPGA implementation of systolic array architecture for matrix multiplication," *International Journal of Computer Applications*, vol. 26, no. 3, pp. 18–22, 2011.

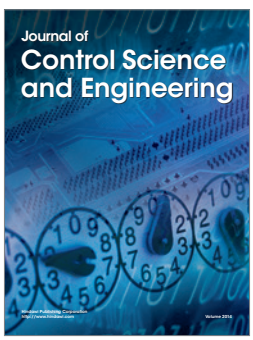
[8] J. Jiang, V. Mirian, K. P. Tang, P. Chow, and Z. Xing, "Matrix multiplication based on scalable macro-pipelined FPGA accelerator architecture," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '09)*, pp. 48–53, Quintana Roo, Mexico, December 2009.

[9] C. Lin, Z. Zhang, N. Wong, and H. So, "Design space exploration for sparse matrix-matrix multiplication on FPGAs," in *Proceedings of the International Conference on Field-Programmable Technology (FPT '10)*, pp. 369–372, Beijing, China, December 2010.

[10] V. Kumar, S. Joshi, S. Patkar, and H. Narayanan, "FPGA based high performance double-precision matrix multiplication," *International Journal of Parallel Programming*, vol. 38, no. 3-4, pp. 322–338, 2010.







Hindawi
Submit your manuscripts at
http://www.hindawi.com