Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 514581, 15 pages
doi:10.1155/2011/514581

Research Article

The Potential for a GPU-Like Overlay Architecture for FPGAs

Jeffrey Kingyens and J. Gregory Steffan

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada M5S 3G4

Correspondence should be addressed to J. Gregory Steffan, steffan@eecg.toronto.edu

Received 3 August 2010; Accepted 27 December 2010

Academic Editor: Aravind Dasu

We propose a soft processor programming model and architecture inspired by graphics processing units (GPUs) that are well-matched to the strengths of FPGAs, namely, highly parallel and pipelinable computation. In particular, our soft processor architecture exploits multithreading, vector operations, and predication to supply a floating-point pipeline of 64 stages via hardware support for up to 256 concurrent thread contexts. The key new contributions of our architecture are mechanisms for managing threads and register files that maximize data-level and instruction-level parallelism while overcoming the challenges of port limitations of FPGA block memories as well as memory and pipeline latency. Through simulation of a system that (i) is programmable via NVIDIA's high-level `Cg` language, (ii) supports AMD's CTM r5xx GPU ISA, and (iii) is realizable on an XtremeData XD1000 FPGA-based accelerator system, we demonstrate the potential for such a system to achieve 100% utilization of a deeply pipelined floating-point datapath.

1. Introduction

As FPGAs become increasingly dense and powerful, with high-speed I/Os, hard multipliers, and plentiful memory blocks, they have consequently become more desirable platforms for computing. Recently there is building interest in using FPGAs as accelerators for high-performance computing, leading to commercial products such as the SGI RASC which integrates FPGAs into a blade server platform, and XtremeData and Nallatech that offer FPGA accelerator modules that can be installed alongside a conventional CPU in a standard dual-socket motherboard.

The challenge for such systems is to provide a programming model that is easily accessible for the programmers in the scientific, financial, and other data-driven arenas that will use them. Developing an accelerator design in a hardware description language such as Verilog is difficult, requiring an expert hardware designer to perform all of the implementation, testing, and debugging required for developing real hardware. Behavioral synthesis techniques—that allow a programmer to write code in a high-level language such as `C` that is then automatically translated into custom hardware circuits—have long-term promise [1–3], but currently have many limitations.

What is needed is a high-level programming model specifically tailored to making the creation of custom FPGA-based accelerators easy. In contrast with the approaches of custom hardware and behavioral synthesis, a more familiar model is to use a standard high-level language and environment to program a processor, or in this case an FPGA-based soft processor. In general, a soft-processor-based system has the advantages of (i) supporting a familiar programming model and environment and (ii) being portable across different FPGA products and families, while (iii) still allowing the flexibility to be customized to the application. Although soft processors themselves can be augmented with accelerators that are in turn created either by hand or via behavioral synthesis, our long-term goal is to develop a soft processor architecture that is more naturally capable of fully utilizing the FPGA.

1.1. A GPU-Inspired System. Another recent trend is the increasing interest in using the Graphics Processing Units (GPUs) in standard PC graphics cards as general-purpose accelerators, including NVIDIA's CUDA, OpenCL, and AMD (ATI)'s Close-to-the-Metal (CTM) [4] programming environments. While the respective strengths of GPUs and FPGAs are different—GPUs excel at floating-point computation, while FPGAs are better suited to fixed-point and nonstandard bit width computations—they are both very well-suited to highly parallel and pipelinable computation. These programming models are gaining traction which can potentially be leveraged if a similar programming model can be developed for FPGAs.

In addition to the programming model, there are also several main architectural features of GPUs that are very desirable for a high-throughput soft processor. In particular, while some of these features have been implemented previously in isolation and shown to be beneficial for soft processors, our research highlights that when implemented in concert, they are key for the design of a high-throughput soft processor.

Multithreading. Through hardware support for multiple threads, a soft processor can tolerate memory and pipeline latency and avoid the area and potential clock frequency costs of hazard detection logic—as demonstrated in previous work for pipelines of up to seven stages and support for up to eight threads [5–7]. In our high-throughput soft processor we essentially avoid stalls of any kind for very deeply pipelined functional units (64 stages) via hardware support for many concurrent threads (currently up to 256 threads).

Vector Operations. A vector operation specifies an array of memory or register elements on which to perform an operation. Vector operations exploit data-level parallelism as described by software, allowing fewer instructions to command larger amounts of computation, and providing a powerful axis along which to scale the size of a single soft processor to improve performance [8, 9].

Predication. To allow program flexibility it is necessary to support control flow within a thread, although any control flow will make it more challenging to keep the datapath fully utilized—hence we support predicated instructions that execute unconditionally, but have no impact on machine state for control paths that are not taken.

Multiple Processors. While multithreading can allow a single datapath to be fully utilized, instantiating multiple processors can allow a design to be scaled up to use available FPGA resources and memory bandwidth [10]. The GPU programming model specifies an abundance of threads, and is agnostic to whether those threads are executed in the multithreaded contexts of a single processor or across multiple processors. Hence the programming model and architecture are fully capable of supporting multiple processors, although we do not evaluate such systems in this work.

1.2. Research Goals. Together, the above features provide the latency tolerance, parallelism, and architectural simplicity required for a high-throughput soft processor. Rather than inventing a new programming model, ISA, and processor architecture to support these features, as a starting point for this research we have ported an existing GPU programming model and architecture to an FPGA accelerator system. Specifically, we have implemented a `system-C` simulation of a GPU-inspired soft processor that (i) is programmable via NVIDIA's high-level `C`-based language called `Cg` [11], (ii) supports an *application binary interface*-(ABI-) based the AMD CTM r5xx GPU ISA [4], and (iii) is realizable on an XtremeData XD1000 development system composed of a dual-socket motherboard with an AMD Opteron CPU and the FPGA module which communicate via a HyperTransport (HT) link.

Our long-term research goal is to use this system to gain insight on how to best architect a soft processor and programming model for FPGA-based acceleration. In this work, through our implementation of the CTM ISA, we demonstrate that our heavily multithreaded GPU-inspired architecture can overcome several key challenges in the design of a high-throughput soft processor for acceleration—namely, (i) the port limitations of on-chip memories in the design of the main register file, (ii) tolerating potentially long latencies to memory, and (iii) tolerating the potentially long latency of deeply pipelined functional units. Furthermore, we envision that several aspects of this architecture can be extended in future implementations to better capitalize on the strengths of FPGAs: we can scale the soft processor in the vector dimension as in previous work [8, 9]; rather than focusing on floating-point computation, we can instead focus on nonstandard bit width computation or other custom functions; finally, we can scale the number of soft processor accelerators via multiprocessor implementations to fully utilize available memory bandwidth.

In this paper we (i) propose a new GPU-inspired architecture and programming model for FPGA-based acceleration based on soft processors that exploit multithreading, vector instructions, predication, and multiple processors; (ii) describe mechanisms for managing threads and register files that maximize parallelism while overcoming the challenge of port limitations of FPGA block memories and long memory and pipeline latencies; (iii) demonstrate that these features, when implemented in concert, result in a soft processor design that can fully utilize a deeply pipelined datapath. This paper extends our previous workshop publication at RAW 2010 [12] in several ways including a more detailed description of support for predication and handling control flow instructions, a more in-depth treatment of related work, nine additional illustrations and figures, and more detailed experimental results.

2. Related Work

2.1. Behavioral Synthesis-Based Compilers. The main challenge of behavioral synthesis algorithms is to identify parallelism in high-level code and generate a hardware circuit to provide concurrent execution of operations. There are many academic and commercial compilers that are based on synthesis to generate a customized circuit for a given task. Examples of such compilers include Impulse Accelerated Technologies Impulse C [13], Altera's C2H [2], Trident [3], Mitrionics Mitrion-C [1], SRC Computer's SRC Carte, ASC [14], and Celoxica's Handel-C [15]. Typically, these tools will compile C-like code to circuit descriptions in HDL which can then be synthesized by standard FPGA

CAD tools for deployment on accelerator systems such as the Xtremedata XD1000. A synthesis-based compiler will exploit data dependences in high-level code to build local, point-to-point routing at the circuit level. Computations can potentially be wired directly from producer to consumer, bypassing a register store for intermediate computations, a step which is required for general purpose processors. This synthesis-based technique of customized circuits can be especially practical for GPU-like computations, as programs are typically short sequences of code. Where the computation is data-flow dominated, it is also possible to exploit data-level parallelism (DLP) by pipelining independent data through the custom circuit. The downsides to a behavioral synthesis approach are that (i) a small change to the application requires complete/lengthy recompilation of the FPGA design, and (ii) the resulting circuit is not easily understood by the user, which can make debugging difficult.

2.2. Soft Uniprocessors. Soft processors are microprocessors instantiated on an FPGA fabric. Two examples of industrial soft processors are the Altera NIOS and the Xilinx Microblaze. As these processors are deployed on programmable logic, they come in various standard configurations and also provide customizable parameters for application-specific processing. The ISA of NIOS soft processors is based on a MIPS instruction set architecture (ISA), while that of Microblaze is a proprietary reduced instruction set computer (RISC) ISA. SPREE [16] is a development tool for automatically generating custom soft processors from a given specification. These soft processor architectures are fairly simple single-threaded processors that do not exploit parallelism other than pipelining.

2.3. Vector Soft Processors. Yu et al. [17] and Yiannacouras et al. [8] have implemented soft vector processors where the architecture is partitioned into independent vector lanes, each with a local vector register file. This technique maps naturally to the dual port nature of FPGA on-chip RAMs and allows the architecture to scale to a large number of vector lanes, where each lane is provided with its own dual port memory. While the success of this architecture relies on the ability to vectorize code, for largely data-parallel workloads, such as those studied in our work, this is not a challenge. Soft vector processors are interesting with respect to our work because we also rely on the availability of data parallelism to achieve performance improvements. However, while vector processors scale to many independent lanes each with a small local register file, our high throughput soft processors focus on access to a single register file. Hence our techniques are independent and therefore make it possible to use both in combination.

2.4. Multithreaded Soft Processors. Fort et al. [5], Dimond et al. [18], and Labrecque and Steffan [6] use multithreading in soft processor designs, and show that it can improve area efficiency dramatically. While these efforts focused on augmenting an RISC-based processor architecture with multithreading capabilities, we focus on supporting a GPU stream processor ISA. As the GPU ISA is required to support floating point-based multiply-add operations, the pipeline depth is much longer. Therefore, we extend the technique here to match the pipeline depth of our functional units. Although we require many more simultaneous threads, the data-parallel nature of the GPU programming model provides an abundance of such threads.

2.5. SPMD Soft Processors. The GPU programming model is only one instance in the general class of Single-Program Multiple-Data (SPMD) programming models. There has been previous work in soft processor systems supporting SPMD. James-Roxby et al. [19] implement a SPMD soft processor system using a collection of Microblaze soft processors attached to a global shared bus. All soft processors are connected to a unified instruction memory as each are executing instructions from the same program. All soft processors are free to execute independently. When a processor finishes executing the program for one piece of data, it will request more work from a soft processor designated to dispatch workloads. While their work focuses on a multi processor system, little attention is paid to the optimization of a single core. This is primarily because the work is focused on SPMD using soft processors as a rapid prototyping environment. While the GPU programming model is scalable to many processors, we focus on the optimization of a single processor instance. The system-level techniques used by James-Roxby et al. [19] such as instruction memory sharing between processors could be applied in a multiprocessor design of our high-throughput soft processors.

2.6. Register File Access in Soft Processors. As instruction-level parallelism increases in a soft processor design, more read and write ports are required to sustain superscalar instruction issue and commit rates. In trying to support the AMD r5xx GPU ISA, we were confronted with the same problem, as this ISA requires 4 read and 3 write accesses from a single register file, each clock cycle, if we are to fully pipeline the processor datapath. One possibility is to implement a multiported register file using logic elements as opposed to built-in SRAMs. For example, Jones et al. [20] implement such a register file using logic elements. However, they show that using this technique results in a register file with very high area consumption, low clock frequency, and poor scalability to a large number of registers. LaForest and Steffan [21] summarize the conventional techniques for building multiported memories out of FPGA block RAMs (replication, banking, and multipumping) and also describe a new technique called the *live value table*. The solution we propose in this paper is a form of banking that exploits the availability of independent threads. Saghir et al. [22] have also used multiple dual-port memories to implement a banked register file, allowing the write-back of two instructions per cycle in cases where access conflicts do not occur; however, they must rely on the compiler to schedule register accesses within a program such that writes are conflict free. We exploit the execution of multiple threads in lock step, allowing us to build conflict-free accesses to a

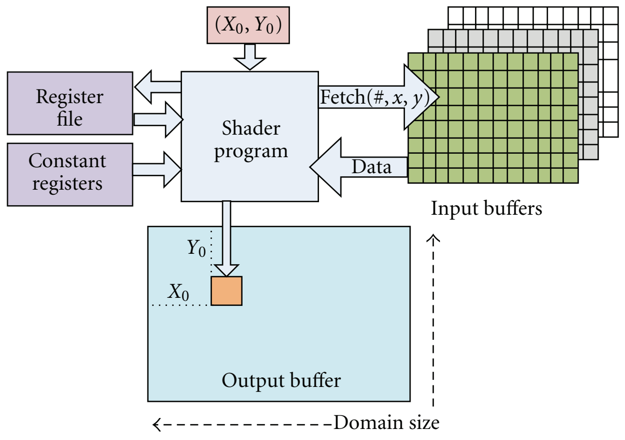

Figure 1: The interaction of a shader program with memories and registers.

banked register file in hardware. While they bank the register file across multiple on-chip memories, we provide banked access both within a single register, through interleaving with a wide memory port, in addition to accessing across multiple memory blocks.

3. System Overview

In this section, we give an overview of our system as well as for GPUs, in particular their shader processors. We also briefly describe the two software interfaces that we use to build our system: (i) NVIDIA's Cg language for high-level programming and (ii) the AMD CTM SDK that defines the application binary interface (ABI) that our soft processor implements.

3.1. GPU Shader Processors. While GPUs are composed of many fixed-function and programmable units, the shader processors are the cores of interest for our work. For a graphics workload, shader processors perform a certain computation on every vertex or pixel in an input stream, as described by a shader program. Since the computation across vertices or pixels is normally independent, shader processors are architected to exploit this parallelism: they are heavily multithreaded and pipelined, with an ISA that supports both vector parallelism as well as predication. Furthermore, there are normally multiple shader processors to improve overall throughput.

Figure 1 illustrates how a shader program can interact with memory: input buffers can be randomly accessed while output is limited to a fixed location for each shader program instance, as specified by an input register. Hence the execution of a shader program implies the invocation of parallel instances across all elements of the output buffer. This separation and limitation for writing memory simplifies issues of data coherence and is more conducive to high-bandwidth implementations.

3.2. The NVIDIA Cg Language. In our system we exploit NVIDIA's Cg [11], a high-level, C-based programming language that targets GPUs. To give a taste of the Cg language, Figure 2(b) shows a sample program written in Cg for elementwise multiplication of two matrices, with the addition of a unit offset. For comparison, ANSI C code is provided for the same routine in Figure 2(a). In Cg, the multadd function defines a computation which is implicitly executed across each element in the output domain, hence there are no explicit for loops for iterating over the output buffer dimensions as there would be in regular C. The dimensions of the buffers are configured prior to execution of the shader program and hence do not appear in the Cg code either.

Looking at the Cg code, there are three parameters passed to the multadd function. First, 2D floating-point coordinates (coord) directly give the position for output in the output buffer and are also used to compute the positions of values in input buffers (i.e., the (X_0,Y_0) input pair shown in Figure 1)—note that a future implementation could remove this limitation to allow many-dimensional coordinates. The second and third parameters (A and B) define the input buffers, associated with a buffer number (TEXUNIT0 and TEXUNIT1) and a memory addressing mode, uniform sampler2D, that in this case tells the compiler to compute addresses same way C computes memory addresses for 2D arrays. Tex2D() is an intrinsic function to read data from an input buffer, and implements the addressing mode specified by its first parameter on the coordinates specified by its second parameter. For this program the values manipulated including the output value are all of type float4, a vector of four 32-bit floating-point values—hence the buffer sizes and addressing modes must account for this.

While Cg allows the programmer to abstract-away many of the details of the underlying GPU ISA, it is evident that an ideal high-level language for general-purpose acceleration would eliminate the remaining graphics-centric artifacts in Cg.

3.3. AMD's CTM SDK. The AMD CTM SDK is a programming specification and tool set developed by AMD to abstract the GPU's shader processor core as a data-parallel accelerator [4, 23], hiding many graphics-specific aspects of the GPU. As illustrated in Figure 3, we use the cgc compiler included in NVIDIA's Cg toolkit to compile and optimize shader programs written in Cg. cgc targets Microsoft pixel shader virtual assembly language (ps3), which we then translate via CTM's amucomp compiler into the AMD CTM application binary interface (ABI) based on the r5xx ISA. The resulting CTM shader program binary is then folded into a *host program* that runs on the regular CPU. The host program interfaces with a low-level *CTM driver* that replaces a standard graphics driver, providing a *computer* interface (as opposed to graphics-based interface) for controlling the GPU. Through driver API calls, the host program running on the main CPU configures several parameters prior to shader program execution, including the base address and sizes of input and output buffers as well as constant register data (all illustrated in Figure 1). The host program also uses the CTM driver to load shader program binaries onto the GPU for execution.

```
float   A[WIDTH][HEIGHT];   // input buffer A
float   B[WIDTH][HEIGHT];   // input buffer B
float   C[WIDTH][HEIGHT];   // output buffer

void multadd(void){
  for(int j=0;j<HEIGHT;j++)
    for(int i=0;i<WIDTH;i++)
      C[i][j] = A[i][j]*B[i][j] + 1.0f;
}
```

(a) C code

```
struct data_out {
  float4 sum : COLOR;
};

data_out
multadd(float2 coord : TEXCOORD0,
        uniform sampler2D A: TEXUNIT0,
        uniform sampler2D B: TEXUNIT1){
  data_out r;
  float4 offset = {1.0f, 1.0f, 1.0f, 1.0f};
  r.sum = tex2D(A,coord)*tex2D(B,coord)+offset;
  return r;
}
```

(b) Cg code

```
multadd:
  TEX r1 r0.rg s1
    // r1 = r0.r + (r0.g*s1.width) + s1.base
  TEX r0 r0.rg s0
    // r0 = r0.r + (r0.g*s0.width) + s0.base
  MAD o0 r1 r0 c0
    // o0 = r1 * r0 + c0 (3 left-most elems)
  mad o0 r1 r0 c0
    // o0 = r1 * r0 + c0 (1 right-most elem)
  END
```

(c) CTM code

Figure 2: An example shader program for elementwise matrix multiplication plus an offset, described in (a) C, (b) Cg, and (c) CTM code.

Figure 2(c) shows the resulting CTM code from the example shader program in Figure 2(b). From left to right, the format of an instruction is *opcode*, *destination*, and *sources*. There are several kinds of registers in the CTM ISA: (i) general-purpose vector registers (r0–r127); (ii) "sampler" registers (s0–s15), used to specify the base address and width of an input buffer (i.e., TEXUNIT0–TEXUNIT15 in Cg code); (iii) constant registers (c0–c255), used to specify constant values; (iv) output registers (o0–o3) that are used as the destination for the final output values which are streamed to the output buffer (shown in Figure 1) when the shader program instance completes. All registers are each a vector of four 32-bit elements where the individual elements of the vector are named r, g, b, and a. Both base registers and constant registers are configured during setup by the CTM driver, but are otherwise read-only.

CTM defines both TEX and ALU instructions. A TEX instruction defines a memory load from an input buffer, and essentially implements the Tex2D() call in Cg. The input coordinates (coord in Cg) are made available in register r0 at the start of the shader program instance. The address is computed from both r0 and a sampler register (i.e., s0). For example, the address for the sources given as r0·rgs1 is computed as r0.r+r0.g∗s1.width+s1.base. All ALU instructions are actually VLIW operation pairs that can be issued in parallel: a three-element vector operation specified in uppercase, followed (on a new line) by a scalar operation specified in lower case. In the example the ALU instruction is a pair of *multiply adds* that specify three-source operands and one destination operand for both the vector (MAD) and scalar (mad) operations. ALU instructions can access any of r0–r127 and c0–c255 as any source operand.

CTM allows many other options that we do not describe here, such as the ability to permute (swizzle) the elements of the vectors after loading from input buffers or before performing ALU operations, and also for selectively masking the elements of destination registers. A complete description of the r5xx ISA and the associated ABI format is available in the CTM specification [23].

In summary, this software flow allows us to support existing shader programs written in `Cg`, and also allows us to avoid inventing our own low-level ISA.

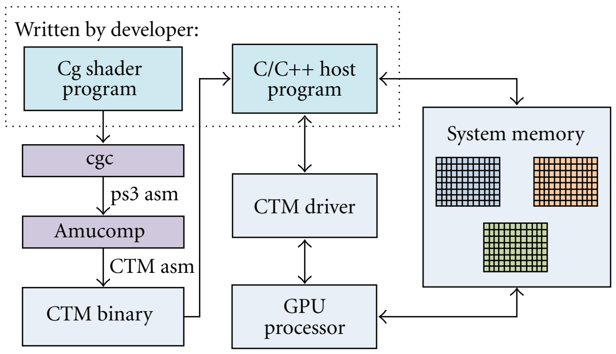

FIGURE 3: The software flow in our system. A software developer writes a high-level `Cg` shader program and a host program. The `Cg` shader program is translated into a CTM binary via the `cgc` and `amucomp` compilers and is then folded into the host program.

4. A GPU-Inspired Architecture

In this section we describe the architecture of our high-throughput soft-processor accelerator, as inspired by GPU architecture. First we describe an overview of the architecture, and explain in detail the components that are relatively straightforward to map to an FPGA-based design. We then describe three features of the architecture that overcome challenges of an FPGA-based design.

4.1. Overview. Figure 4 illustrates the high-level architecture of the proposed GPU-like accelerator. Our architecture is designed specifically to interface with a HyperTransport (HT) master and slave, although interfacing with other interconnects is possible. The following describes three important components of the accelerator that are relatively straightforward to map to an FPGA-based design.

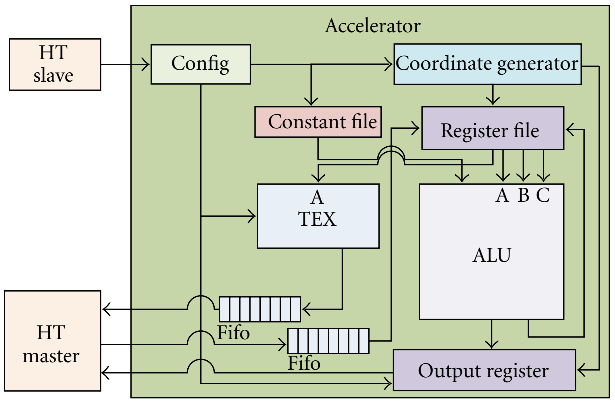

FIGURE 4: Overview of a GPU-like accelerator, connected to a Hypertransport (HT) master and slave.

Coordinate Generation. As described in Section 3 and by the CTM specification, a shader program instance is normally parameterized entirely by a set of input coordinates which range from the top-left to the bottom-right of the compute domain. The coordinate generator is configured with the definition of the compute domain and generates streams of coordinates which are written into the register file (register `r0`) for shader program instances to read—replacing outer-looping control flow in most program kernels.

TEX and ALU Datapaths. TEX instructions, which are essentially loads from input buffers in memory, are executed by the TEX datapath. Once computed based on the specified general-purpose and sampler registers, the load address is packaged as an HT read request packet and sent to the HT core—unless there are already 32 in-flight previous requests in which case the current request is queued in a FIFO buffer. When a request is satisfied, any permutation operations (as described in Section 3.3) are applied to the returned data and the result is written back to the register file. The CTM ISA also includes a method for specifying that an instruction depends on the result of a previous memory request (via a special bit). Each TEX instruction holds a semaphore that is cleared once its result is written back to the register file—which signals any awaiting instruction to continue. ALU instructions are executed by the ALU datapath, and their results can be written to either the register file or the output register.

Predication and Control Flow. Conditional constructs such as *if* and *else* statements in `Cg` are supported in CTM instructions via predication. There is one predicate bit per vector lane that can be set using the boolean result from one of many comparison operations (e.g., >, <=, ==, ! =). Subsequent ALU instructions can then impose a write mask conditional on the values of these bits. More complex control flow constructs such as *for* loops and subroutines are supported via a control flow instructions that provide control of hardware-level call/return stacks, and branch and loop-depth hierarchies; space limitations prevent us from fully-explaining control flow instructions here, hence we refer the reader to the CTM specification [23] for further details.

Output. Similar to input buffers, the base addresses and widths of the output buffers are preconfigured by the CTM driver in advance (in the registers o0–o3). When a shader program instance completes, the contents of the output registers are written to the appropriate output buffers in memory: the contents of the output registers are packaged into an HT write request packet, using an address derived from one of the output buffer base addresses and the original input coordinates (from the coordinate generator). Write requests are *posted*, meaning that there is no response packet and hence no limit on the maximum number of outstanding writes.

4.2. Tolerating Limited Memory Ports. In Figure 4 we observe that there are a large number of ports feeding into and out of the central register file (which holds r0–r127). One of the biggest challenges in high-performance soft processor design is the design of the register file: it must tolerate the port limitations of FPGA block memories that are normally limited to only two ports. To fully pipeline the ALU and TEX datapaths, the central register file for our GPU-inspired accelerator requires four read and three write ports. If we attempted a design that reads all of the ALU and TEX source operands (four of them) of a single thread in a single cycle, we would be required to have replicated copies of the register file across multiple block memories to have enough ports. However, this solution does not provide more than one write port, since each replicant would have to use one port for reading operands and the other port for broadcast-writing the latest destination register value (i.e., being kept up-to-date with one write every cycle).

We solve this problem by exploiting the fact that all threads are executing different instances of the same shader program: all threads will execute the exact same sequence of instructions, since even control flow is equalized across threads via predication. This symmetry across threads allows us to group threads into batches and execute the instructions of batched threads in lock-step. This lock-step execution in turn allows us to *transpose* the access of registers to alleviate the ports problem.

Rather than attempting to read all operands of a thread each cycle, we instead read a single operand across many threads per cycle from a given block memory and do this across separate block memories for each component of the vector register. Table 1 illustrates how we schedule register file accesses in this way for batches of four threads each that are decoding only ALU instructions (for simplicity). Since there are three operands to read for ALU instructions this adds a three-cycle decode latency for such instructions. However, in the steady-state we can sustain our goal of the execution of one ALU instruction per cycle, hence this latency is tolerable. This schedule also leaves room for another read of an operand across threads in a batch. Ideally we would be able to issue the register file read for a TEX instruction during this slot, which would allow us to fully utilize the central register file, ALU datapath and TEX datapath: every fourth cycle we would read operands for a batch of four threads for a TEX instruction, then be able to issue a TEX instruction for each of those threads over the next four cycles. We give the name transpose to this technique of scheduling register accesses.

This transposed register file design also eases the implementation of write ports. In fact, the schedule in Table 1 uses only one read port per block memory, leaving the other port free for writes. From the table we see that ALU instructions will generate at most one register write across threads in a batch every four cycles. There are two other events which result in a write to the central register file: (i) a TEX instruction completes, meaning that the result has returned from memory and must be written-back to the appropriate destination register; (ii) a shader program instance completes for a batch of threads and a new batch is configured, so that the input coordinates must be set for that new batch (register `r0`). These two types of register write are performed immediately if the write port is free, otherwise they are queued until a subsequent cycle.

Table 1: The schedule of operand reads from the central register file for batches of four threads (T0–T3, T4–T7, etc.) decoding only ALU instructions. An ALU instruction has up to three vector operands (A, B, C) which are read across threads in a batch over three cycles. In the steady state this schedule can sustain the issue of one ALU instruction from every cycle.

Clock cycle	Inst phase	Register file read	ALU ready
0	ALU_0	ALU:A(T0,T1,T2,T3)	—
1	ALU_1	ALU:B(T0,T1,T2,T3)	—
2	ALU_2	ALU:C(T0,T1,T2,T3)	—
3	—	—	T0
4	ALU_0	ALU:A(T4,T5,T6,T7)	T1
5	ALU_1	ALU:B(T4,T5,T6,T7)	T2
6	ALU_2	ALU:C(T4,T5,T6,T7)	T3
7	—	—	T4
8	ALU_0	ALU:A(T8,T9,T10,T11)	T5
9	ALU_1	ALU:B(T8,T9,T10,T11)	T6
10	ALU_2	ALU:C(T8,T9,T10,T11)	T7
11	. . .	. . .	. . .

4.3. Avoiding Pipeline Bubbles. In the previous section we demonstrated that a transposed register file design can allow the hardware to provide the register reads and writes necessary to sustain the execution of one ALU instruction every cycle across threads. However, there are three reasons why issuing instructions to sustain such full utilization of the datapaths is a further challenge. The first reason is as follows. In the discussion of Table 1 we described that the ideal sequence of instructions for fully utilizing the ALU and TEX datapaths is an instruction stream which alternates between ALU and TEX instructions. This is very unlikely to happen naturally in programs, and the result of other nonideal sequences of instructions will be undesirable bubbles in the two datapaths. The second reason, as shown in Figure 5, is that the datapath for implementing floating-point operations such as multiply add (`MAD`) and dot product (`DOT3`, `DOT4`) instructions is very long and deeply pipelined (64 clock cycles): since ALU instructions within a thread will often have register dependences between them, this can prevent an ALU instruction from issuing until a previous ALU instruction completes. This potentially long stall will also result in unwanted bubbles in the ALU datapath.

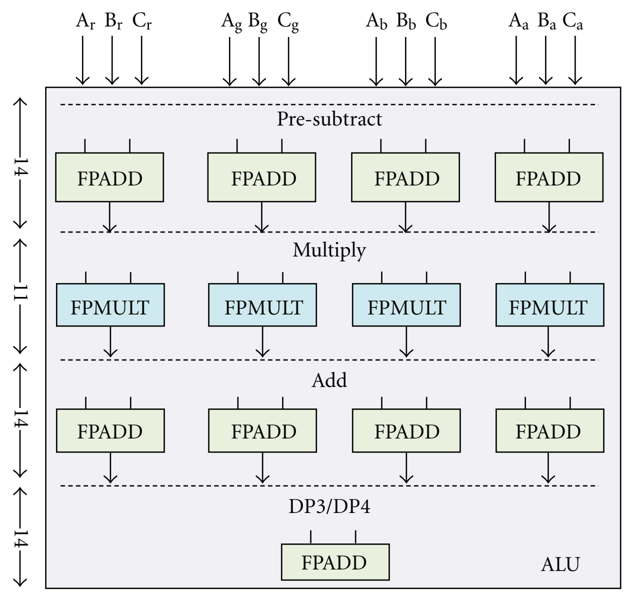

Figure 5: The floating point units in a datapath that supports MADD, DP3, and DP4 ALU instructions. The pipeline latency of each unit is shown on the left (for Altera floating point IP cores), and the total latency of the datapath is 53 cycles without accounting for extra pipeline stages for multiplexing between units.

Table 2: The schedule of operand reads from the central register file for batches of four threads (T0–T3, T4–T7, etc.) decoding both ALU and TEX instructions. TEX instructions require only one source operand, hence we can read source operands for four threads in a single cycle.

Clock cycle	Inst phase	Register file read	ALU ready	TEX ready
0	ALU_0	ALU:A(T0,T1,T2,T3)	—	—
1	ALU_1	ALU:B(T0,T1,T2,T3)	—	—
2	ALU_2	ALU:C(T0,T1,T2,T3)	—	—
3	TEX	TEX:A(T0,T1,T2,T3)	T0	—
4	ALU_0	ALU:A(T4,T5,T6,T7)	T1	T0
5	ALU_1	ALU:B(T4,T5,T6,T7)	T2	T1
6	ALU_2	ALU:C(T4,T5,T6,T7)	T3	T2
7	TEX	TEX:A(T4,T5,T6,T7)	T4	T3
8	ALU_0	ALU:A(T8,T9,T10,T11)	T5	T4
9	ALU_1	ALU:B(T8,T9,T10,T11)	T6	T5
10	ALU_2	ALU:C(T8,T9,T10,T11)	T7	T6
11	...	...	...	...

The third reason is that TEX instructions can incur significant latency since they load from main memory; since an ALU instruction often depends on a previous TEX instruction for a source operand, the ALU instruction would have to stall until the TEX instruction completes.

We address all three of these problems by storing the contexts of multiple batches in hardware, and dynamically switching between batches every cycle. We capitalize on the fact that all threads are independent across batches as well as within batches, switching between batches to (i) choose a batch with an appropriate next instruction to match the available issue phase (TEX or ALU) and (ii) to hide both pipeline and memory latency. This allows us to potentially fully utilize both the ALU and TEX datapaths, provided that ALU and TEX instructions across all batch contexts are ready to be issued when required. Specifically, to sustain this execution pattern we generally require that the ratio of ALU to TEX instructions be 1.0 or greater: for a given shader program if TEX instructions outnumber ALU instructions then in the steady state this alone could result in pipeline bubbles. Storing the contexts (i.e., register file state) of multiple batches is relatively straightforward: it requires only growing the depth of the register file to accommodate the additional registers—although this may require multiple block memories to accomplish. In the next section we describe the implementation of batch issue logic in greater detail.

4.4. Control Flow. The thread batching and scheduling solutions above present problems for the control flow instruction, FLW. The first problem of finding time to schedule the execution of such instructions is easily solved. Column 2 in Table 2 shows two cycles each period where the ALU is fetching operands B and C. As the batch scheduler and instruction issue logic is idle during this time, the hardware can be used to schedule an FLW instruction. Since FLW requires neither a read or write to the register file, no structural hazards arise.

The second problem is how to resolve diverging control flow. Diverging control flow is when threads within a batch decide to take alternate branch paths. It turns out, the r5xx ISA has encoded support within the FLW instruction to resolve this specific problem. This is because GPUs use the same technique to resolve diverging control path when executing multiple threads using SIMD hardware. The way these instructions are handled is through the hardware management of thread states which are not visible to the programmer. These thread states are manipulated by the control flow instructions, depending on the previous thread states (the state before an FLWinstruction is executed), the evaluation of the branch condition, and a resolution function when threads disagree. Much of this hardware level management is considered on a case-by-case basis. For example, as threads execute over an else instruction the active state of each thread is flipped. This requires all threads to execute serially through all branch paths and mask register writes when they are inactive. Fung et al. [24] have explored optimizations of this technique in more detail. For more details of how the r5xx ISA programs and handles control flow, see the CTM specification [23].

5. Implementation

This section describes our work to implement our GPU-inspired accelerator on the XtremeData platform. After an overview of the XtremeData XD1000 and how we map the CTM system to it, we describe the low-level implementation of the two key components of our accelerator: the central register file and the batch issue logic.

5.1. The XtremeData XD1000. As illustrated in Figure 6, the XtremeData XD1000 is an accelerator module that contains an Altera Stratix II EPS180 FPGA, and that plugs into a standard CPU socket on a multisocket AMD Opteron motherboard. IP cores are available for the FPGA which allow access to system memory via Hypertransport (HT) that provides a single physical link per direction, each of which is a 16-bit-wide 400 MHz DDR interface and can transfer 1.6 GB/sec. The host CPU treats the XD1000 as an endpoint that is configured by a software driver to respond to a memory-mapped address range using the HT slave interface, similar to other regular peripherals. The FPGA application can also initiate DMA read and write transactions to system memory by constructing and sending HT request packets, providing efficient access to memory without involving the CPU. In our work we extend the XtremeData system to conform to the CTM interface by (i) adding a driver layer on top of the XD1000 driver, and (ii) by memory-mapping our accelerator's hardware configuration state registers and instruction memory to the HT slave interface. Instruction memory resides completely on-chip and stores up to 512 instructions—the limit currently defined by CTM. Each instruction is defined by the ABI to be 192 bits, hence the instruction store requires three M4K RAM blocks. The RAM blocks have two ports: one is configured as a write port that is connected directly to the configuration block so that the CTM driver can write instructions into it; the other is configured as a read port to allow the accelerator to fetch instructions. The CTM driver initializes the accelerator with the addresses of the start and end instruction of the shader program and initiates execution by writing to a predetermined address.

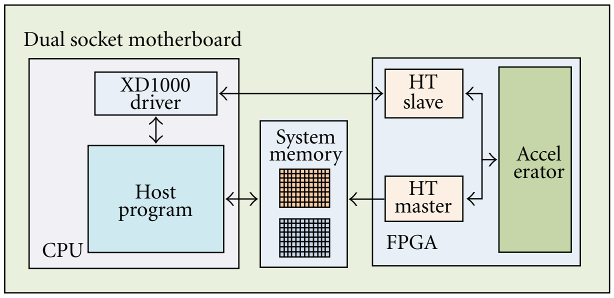

Figure 6: An XtremeData system with a XD1000 module.

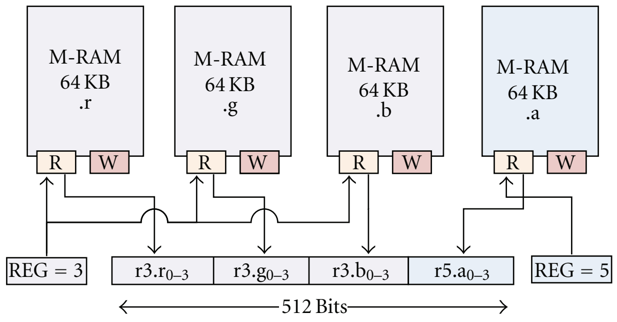

Figure 7: Mapping our register file architecture to four Stratix II's 64 KB M-RAM blocks. The read circuitry shows an example where we are reading operands across threads in a batch for a vector/scalar ALU instruction pair (VLIW): `r3` as an operand for the vector instruction and `r5` as an operand for the scalar instruction. While not shown, register writes are implemented similarly.

5.2. Central Register File. While our transposed design allows us to architect a high-performance register file using only two ports, the implementation has the additional challenges of (i) supporting the vast capacity required, and (ii) performing the actual transposition. Each batch is composed of four threads that each require up to 128 registers, where each register is actually a vector of four 32-bit elements. We therefore require the central register file to support 8 KB of on-chip memory per batch. For example, as illustrated in Figure 7, 32 batches would require 256 KB of on-chip memory, which means that we must use four of the 64 KB M-RAM blocks available in the Stratix II chip in the XD1000 module. Figure 8 shows the circuit we use to transpose the operands read across threads in a batch for ALU instructions so that the three operands for a single instruction are available in the same cycle: a series of registers buffer the operands until they can be properly transposed.

5.3. Batch Issue Logic. As described previously in Section 4.3, to ensure that the ALU datapath is fully utilized our soft processor schedules instructions to issue across batches. For a given cycle, we ideally want to find either an ALU or TEX instruction to issue. Figure 9 shows the circuit that performs this batch scheduling, for an example where we want to find an ALU instruction to issue. We can trivially compare the desired next instruction type (ALU in this case) with the actual next instruction type for each batch as recorded in the batch state register, since this information about the next instruction is encoded in each machine instruction (as defined by the CTM ABI). As shown in the figure, we take the set of boolean signals that indicate which batches have the desired next instruction ready to issue and rotate them, then feed the rotated result into a priority encoder that gives the batch number to issue. The rotation is performed such that the previously selected batch is in the lowest-priority position. In the example, we rotate such that the signal for batch 2 is in the lowest-priority right most position, and the priority encoder hence chooses batch 0 as the first batch with a ready ALU instruction. For a GPU-like programming model where all threads are executing the same sequence of instructions, this is sufficient to ensure fairness and forward progress. The batch issue logic can be pipelined with a total budget of four cycles for the circuit, hence during a second cycle the the batch number is used to index the context memory to read the program counter value for that batch, and during a third cycle the program counter value is used to index the instruction memory for the appropriate instruction.

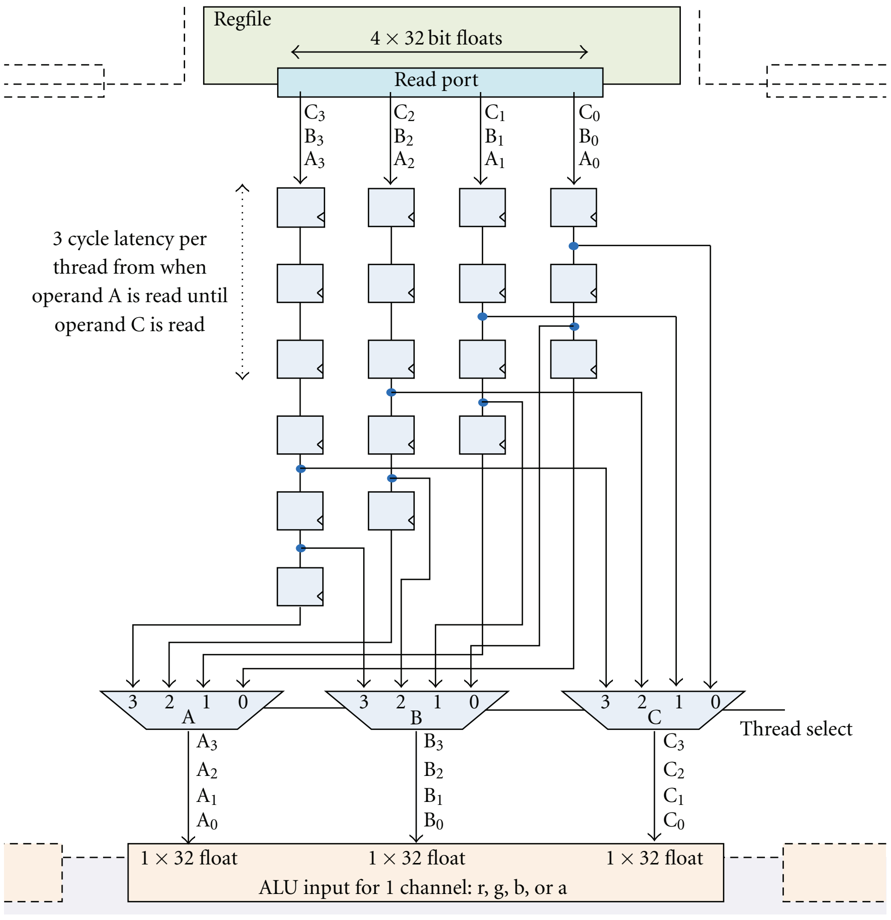

Figure 8: A circuit for transposing the thread-interleaved operands read from the central register file into a correctly ordered sequence of operands for the ALU datapath.

6. Measurement Methodology

In this section we describe the system simulation and benchmark applications that we use to measure the performance of our GPU-inspired soft processor implementation.

6.1. System Simulation. We have developed a complete simulation framework in `SystemC` to measure the ALU utilization and overall performance of workloads on our GPU-like soft processor. Note that we verify the functional accuracy of our simulation by comparing with the outputs of the CTM programs running on the ATI RV570 GPU (on an ATI Radeon x1950 Pro graphics card).

Clock Frequency. Since we do not have a full RTL implementation of our soft processor, we instead assume a system clock frequency of 100 MHz. We choose this frequency to match the 100 MHz HT IP core, which in turn is designed to match a 4x division of the physical link clock frequency (i.e., 400 MHz). We feel that this clock frequency is achievable since (i) other soft processor designs easily do so for Stratix II FPGAs such as the NIOS II/f which executes up to 220 MHz, and (ii) the GPU programming model and abundance of threads allows us to heavily pipeline all components in our design to avoid any long-latency stages.

HyperTransport. Our simulation infrastructure faithfully models the bandwidth and latency of the HT links between the host CPU and the FPGA on the XD1000 platform. We limit the number of outstanding read requests supported by the HT master block to 32, as defined by the HT specification; additional requests are queued. We compute the latency of an HT read request as a sum of the individual latencies of the subcomponents involved. We assume that our soft processor is running at 100 MHz as described above, and that the memory specification is the standard DDR-333 (166 MHz Bus) SDRAM that comes with the XD1000 system. We assume a constant SDRAM access latency of 51 ns, while a constant latency is of course unrealistic; since it contributes only 17% of total latency, we are confident that modeling the small fluctuations of this latency would not significantly impact our results. The latencies of the HT IP core (both input and output paths) were obtained from Slogsnat et al. [25], and the latencies for the the host HT controller, DDR controller, and DDR access were obtained from Holden [26]. Our HyperTransport model is somewhat idealized since we

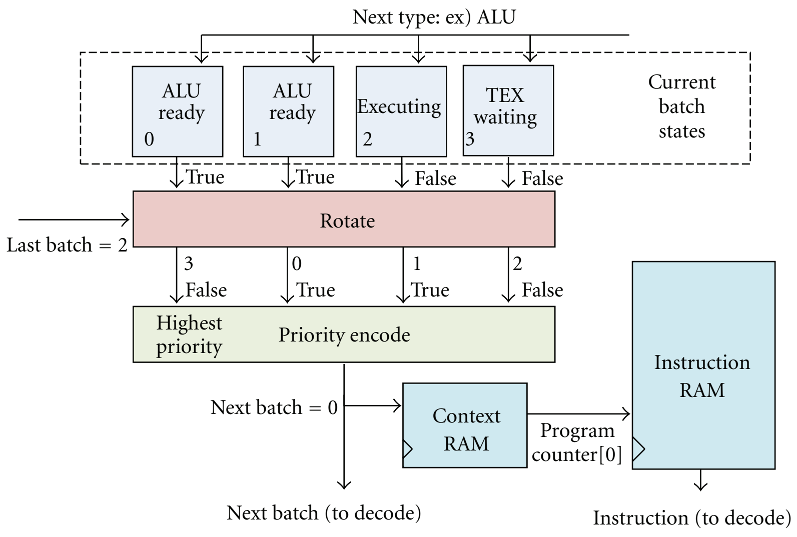

Figure 9: Batch issue logic for hardware managing 4 batch contexts.

Table 3: A breakdown of how each stage of an HT memory request contributes to overall access latency.

Action	Stage	Latency (ns)
Request	(1) FPGA HT IP core	70
	(2) host HT controller	32
SDRAM	(1) access and data fetch	51
Response	(1) host builds response packet	12
	(2) host HT controller	30
	(3) FPGA HT IP core	110
	Total Latency	*305*

do not account for possible HT errors nor contention by the host CPU for memory (Table 3).

Cycle-Accurate Simulation. Our simulator is cycle accurate at the block interfaces shown in Figure 4. For each block we estimate a latency based on the operations and data types present in a behavioral C code implementation. We assume that the batch issue logic shown previously in Figure 9 is fully pipelined, allowing us to potentially sustain the instruction issue schedule shown previously in Table 1.

6.2. Benchmarks. Since our system is compatible with the interface specified by CTM we can execute existing CTM applications, including Cg applications, by simply relinking the CTM driver to our simulation infrastructure. We evaluate our system using the following three applications that have a variety of instruction mixes and behaviors. Note that in our work so far we have not observed any applications with the potentially problematic instruction mix of more TEX instructions than ALU instructions.

Matmatmult. Matmatmult is included with the CTM SDK as CTM assembly code and performs dense matrix-matrix multiplication based on the work of Fatahalian et al. [27]. We selected this application because of its heavy use of TEX instructions to access row and column vectors of an input matrix: the ratio of ALU to TEX instructions for Matmatmult is 2.25.

Sgemm. Sgemm computes $C_{\text{new}} = \alpha(A \cdot B) + \beta C_{\text{old}}$ and represents a core routine of the BLAS math library, and was also included with the CTM SDK as CTM assembly code. Sgemm also makes heavy use of TEX instructions to access two input matrices. The ratio of ALU to TEX instructions for Sgemm is 2.56.

Photon. Photon is a kernel from a Monte Carlo radiative heat transfer simulation, included with the open-source Trident [3] FPGA compiler. We ported this application by hand to Cg such that each instance of the resulting shader program performs the computation for a single particle, and input buffers store previous particle positions and other physical quantities. We selected this benchmark to be representative of applications with higher ratios of ALU to TEX instructions: for Photon it is exactly 4.00.

7. Utilization and Performance

Our foremost goal is to fully utilize the ALU datapath. In this section we measure the ALU datapath utilization for several configurations of our architecture and also measure the impact on performance of increasing the number of hardware batch contexts. Recall that an increasing number of batches provide a greater opportunity for fully utilizing the pipeline and avoiding bubbles by scheduling instructions to issue across a larger number of threads.

Figure 10 shows ALU utilization assuming the 8-bit HT interface provided with the XD1000 system, for a varying number of hardware batch contexts—from one to 64 batches. Since each batch contains four threads, this means that we support from four to 256 threads. We limit the number of

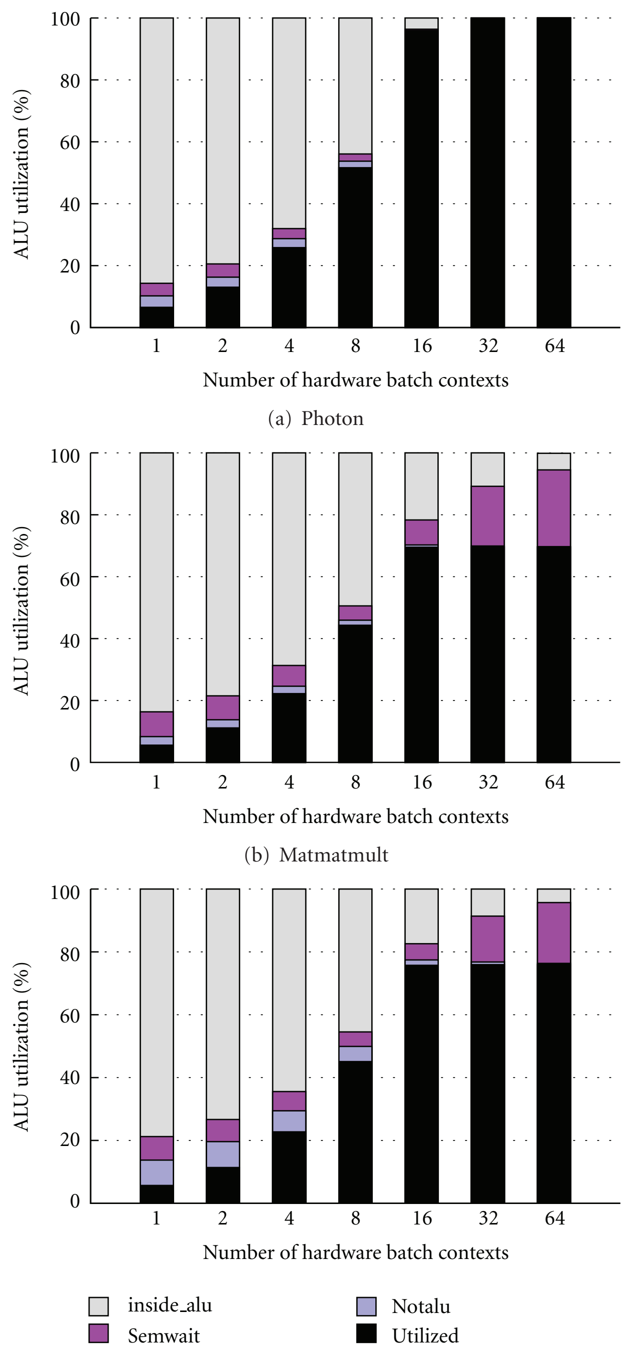

Figure 10: ALU datapath utilization for the 8-bit HT interface provided with the XD1000 system.

batch contexts to 64 because this design includes a central register file that consumes 512 KB of on-chip memory, and thus eight of the nine M-RAMs available in a Stratix II FPGA (64 KB each). In the figure we plot ALU utilization (*utilized*) as the fraction of all clock cycles when an ALU instruction was issued. We also break down the ALU idle cycles into the reasons why no ALU instruction from any batch could be issued (i.e., averaged across all batches contexts). In particular, we may be unable to issue an ALU instruction for a given batch for one of the following three reasons.

Semwait. The next instruction is an ALU instruction, but it is waiting for a memory semaphore because it depends on an already in-flight TEX instruction (memory load).

Inside_ALU. The next instruction is a ready-to-issue ALU instruction, but there is already a previous ALU instruction executing for that batch: since there is no hazard detection logic, a batch must conservatively wait until any previous ALU instruction from that batch completes before issuing a new one, to ensure that any register dependences are satisfied.

NotALU. The next instruction is not an ALU instruction.

From the figure we observe that when only one hardware batch context is supported the ALU datapath is severely underutilized (less than 10% utilization), and the majority of the idle cycles are due to prior ALU instructions in the ALU pipeline (*inside_ALU*). Utilization steadily improves for all three benchmarks as we increase the number of hardware batch contexts up to 16 batches, at which point Matmatmult and Sgemm achieve utilization of 70% and 75%, respectively. However, neither Matmatmult nor Sgemm benefit from increasing further to 32 batches: in both cases waiting for memory is the bottleneck (*Semwait*), indicating that both applications have consumed available memory bandwidth. Similarly, increasing even further to 64 batches yields no improvement, with the memory bottleneck becoming more pronounced. In contrast, for Photon the increase from 16 to 32 batches results in near perfect utilization of the ALU datapath; correspondingly, the increase from 32 to 64 batches cannot provide further benefit. Intuitively, Photon is able to better utilize the ALU datapath because it has a larger ratio of ALU to TEX instructions (four to one).

While the HT IP core provided for the XD1000 is limited to an 8-bit HT interface, the actual physical link connecting the FPGA and CPU is 16 bits. Since memory appears to be a bottleneck limiting ALU utilization, we investigate the impact of a 16-bit HT link such as the one described in [25] as shown in Figure 11. For this improved system we observe that the memory bottleneck is sufficiently reduced to allow full utilization of the ALU datapath for all three benchmarks when 32 hardware batch contexts are supported. In turn, this implies that support for 64 or more hardware batch contexts remains unnecessary. The fact that 32 batches seems sufficient makes intuitive sense since 32 batches comprises 128 threads, while the ALU datapath pipeline is 64 cycles deep and thus requires only that many ALU instructions to be fully utilized—more deeply pipelined ALU functional units would likely continue to benefit from increased contexts.

While maximizing ALU datapath utilization as our overall goal, it is also important to understand the impact of increasing the number of hardware batch contexts on performance. Figure 12. shows speedup relative to a single hardware batch context for both 8-bit and 16-bit HT interfaces. Interestingly, speedup is perfectly linear for between two and eight contexts for all benchmarks and both HT designs, but for 16 and more contexts speedup is sublinear.

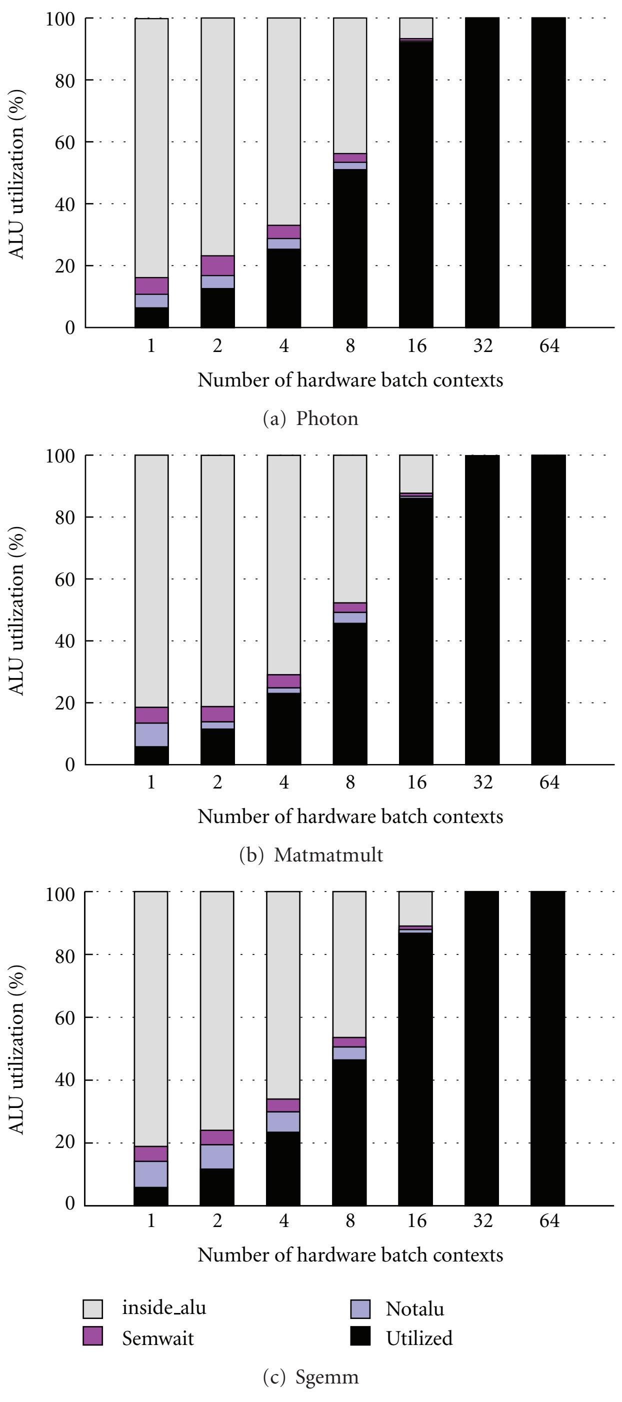

Figure 11: ALU datapath utilization for a 16-bit HT interface.

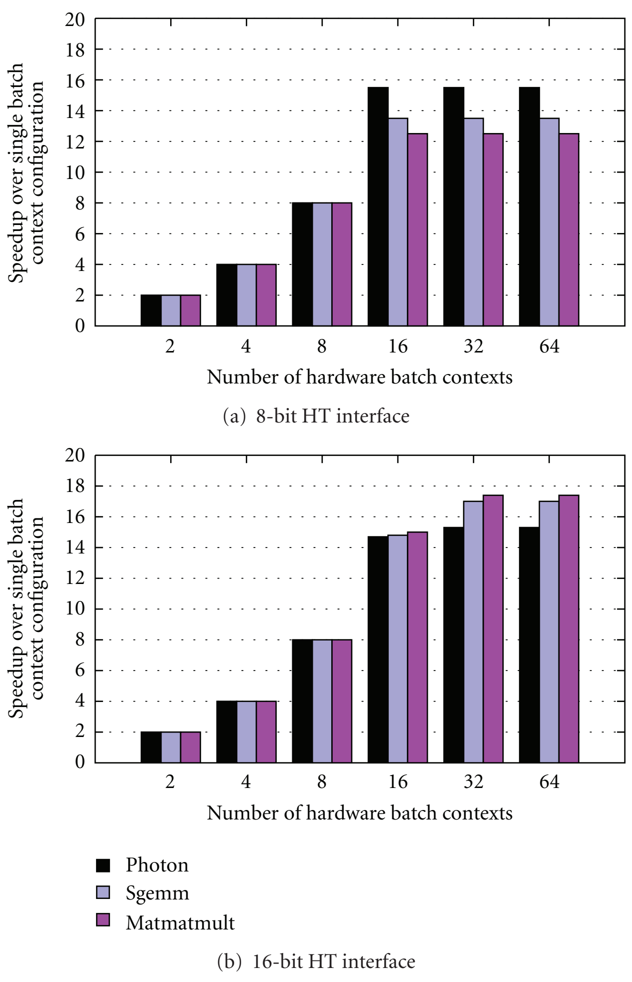

Figure 12: Speedup versus a single hardware batch context for (a) 8-bit and (b) 16-bit HT interfaces.

For the 8-bit HT interface performance does not improve beyond 16 contexts, while for the 16-bit HT interface 32 contexts provide an improvement but 64 contexts do not. Looking at the 16-bit HT interface for 32 contexts, we see that each benchmark speedup is inversely related to the ratio of ALU to TEX instructions: applications with a smaller fraction of TEX instructions benefit less from the latency-tolerance provided by a larger number of contexts. Photon benefits the least and has a ratio of 4.00, followed by Sgemm that has a ratio of 2.56; Matmatmult benefits the most and has a ratio of 2.25.

7.1. Reducing the Register File. While we have demonstrated that 32 hardware batch contexts is sufficient to achieve near perfect ALU datapath utilization, as shown in Section 5.2 this is quite costly, requiring four M-RAMs on the Altera Stratix II. While this may not be an issue if a single accelerator is the only design on the chip, if there are other components or multiple accelerators then storing all of the batch states would be a problem. However, most of the 128 general-purpose vector registers per thread defined by CTM will not be used for many applications. In our architecture it is straightforward to reduce the number of registers supported by a power of two to reduce the total memory requirements for the central register file. For example, Photon, Matmatmult, and Sgemm each use only 4, 15, and 21 general-purpose registers, hence the proposed customization would reduce the size of the central register file by 32x, 8x, and 4x, respectively; for Photon this would instead allow us to build the central register file using only 16 of the much smaller M4 K memories.

7.2. Summary. These results indicate that even for a deeply pipelined ALU of 53 clock cycles we are able to fully utilize this datapath by interleaving the execution of instructions from different batches. This is made possible by the abundance of independent threads provided by the data-parallel GPU programming model.

8. Conclusions

We have presented a GPU-inspired soft processor that allows FPGA-based acceleration systems to be programmed using high-level languages. Similar to a GPU, our design exploits multithreading, vector operations, and predication to enable the full utilization of a deeply pipelined datapath. The GPU programming model provides an abundance of threads that all execute the same instructions, allowing us to group threads into batches and execute the threads within a batch in lock-step. Batched threads allow us to (i) tolerate the limited ports available in FPGA block memories by transposing the operand reads and writes of instructions within a batch and (ii) to avoid pipeline bubbles by issuing instructions across batches. Through faithful simulation of a system that is realizable on an XtremeData XD1000 FPGA-based acceleration platform we demonstrate that our GPU-inspired architecture is indeed capable of fully utilizing a 64-stage ALU datapath when 32 batch contexts are supported in hardware. The long-term goal of this research is to discover new high-level programming models and architectures that allow users to fullyexploit the potential of FPGA-based acceleration platforms; we believe that GPU-inspired programming models and architectures are a step in the right direction.

References

[1] J. Koo, D. Fernandez, A. Haddad, and W. Gross, "Evaluation of a high-level-language methodology for high-performance reconfigurable computers," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '07)*, pp. 30–35, July 2007.

[2] D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 45–54, April 2006.

[3] J. L. Tripp, K. D. Peterson, C. Ahrens, J. D. Poznanovic, and M. B. Gokhale, "Trident: an FPGA compiler framework for floating-point algorithms," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 317–322, August 2005.

[4] J. Hensley, "AMD CTM overview," in *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '07)*, ACM, August 2007.

[5] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for SoPC area reduction," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 131–140, April 2006.

[6] M. Labrecque and J. G. Steffan, "Improving pipelined soft processors with multithreading," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 210–215, August 2007.

[7] R. Moussali, N. Ghanem, and M. A. R. Saghir, "Supporting multithreading in configurable soft processor cores," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '07)*, pp. 155–159, October 2007.

[8] P. Yiannacouras, J. G. Steffan, and J. Rose, "Vespa: portable, scalable, and flexible fpga-based vector processors," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '08)*, 2008.

[9] J. Yu, G. Lemieux, and C. Eagleston, "Vector processing as a soft-core CPU accelerator," in *Proceedings of the 16th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '08)*, pp. 222–231, February 2008.

[10] M. Labrecque, P. Yiannacouras, and J. G. Steffan, "Scaling soft processor systems," in *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, pp. 195–205, April 2008.

[11] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: a system for programming graphics hardware in a c-like language," in *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '03)*, pp. 896–907, ACM, New York, NY, USA, 2003.

[12] J. Kingyens and J. G. Steffan, "A GPU-inspired soft processor for high-throughput acceleration," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, April 2010.

[13] "Developing fpga coprocessors for performance-accelerated spacecraft image processing," *Xcell Journal Second Quarter*, pp. 22–26, 2008.

[14] O. Mencer, "ASC: a stream compiler for computing with FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 9, Article ID 1673737, pp. 1603–1617, 2006.

[15] I. Page, "Closing the gap between hardware and software: hardware-software cosynthesis at Oxford," in *Proceedings of the IEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems*, pp. 201–211, February 1996, Digest no: 1996/036.

[16] P. Yiannacouras, J. Rose, and J. Gregory Steffan, "The microarchitecture of FPGA-based soft processors," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '05)*, pp. 202–212, New York, NY, USA, 2005.

[17] J. Yu, G. Lemieux, and C. Eagleston, "Vector processing as a soft-core CPU accelerator," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '08)*, pp. 222–231, ACM, New York, NY, USA, 2008.

[18] R. Dimond, O. Mencer, and W. Luk, "Application-specific customisation of multi-threaded soft processors," *IEE Proceedings: Computers and Digital Techniques*, vol. 153, no. 3, pp. 173–180, 2006.

[19] P. James-Roxby, P. Schumacher, and C. Ross, "A single program multiple data parallel processing platform for FPGAs," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04)*, pp. 302–303, April 2004.

[20] A. K. Jones, R. Hoare, I. S. Kourtev et al., "A 64-way VLIW/SIMD FPGA architecture and design flow," in *Proceedings of the 11th IEEE International Conference on Electronics, Circuits and Systems (ICECS '04)*, pp. 499–502, December 2004.

[21] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for FPGAs," in *Proceedings of the 18th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '10)*, pp. 41–50, February 2010.

[22] M. A. R. Saghir, M. El-Majzoub, and P. Akl, "Datapath and isa customization for soft vliw processors," in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGA (ReConFig '06)*, pp. 1–10, September 2006.

[23] M. Peercy, M. Segal, and D. Gerstmann, "A performance-oriented data parallel virtual machine forgpus," in *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '06)*, p. 184, ACM, New York, NY, USA, 2006.

[24] W. W .L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of the 40th Annual International Symposium on Microarchitecture (MICRO '07)*, pp. 407–418, IEEE Computer Society, Washington, DC, USA, 2007.

[25] D. Slogsnat, A. Giese, and U. Brüning, "A versatile, low latency HyperTransport core," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '07)*, pp. 45–52, ACM, New York, NY, USA, 2007.

[26] B. Holden, *Latency Comparison between HyperTransport and PCI-Express In Communications Systems*, World Wide Web Electronic Publication, 2006.

[27] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms formatrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 133–137, ACM, New York, NY, USA, 2004.