A GPU-INSPIRED SOFT PROCESSOR FOR HIGH-THROUGHPUT ACCELERATION

Jeffrey Kingyens and J. Gregory Steffan

Department of Electrical and Computer Engineering, University of Toronto
{kingyen,steffan}@eecg.toronto.edu

ABSTRACT

There is building interest in using FPGAs as accelerators for high-performance computing, but existing systems for programming them are so far inadequate. In this paper 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 and vector operations 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) supports AMD's CTM r5xx GPU ISA [1], and (ii) is realizable on an XtremeData XD1000 FPGA-based accelerator system, we demonstrate that our soft processor can achieve 100% utilization of the 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 [2–4], 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 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 and AMD (ATI)'s Close-to-the-Metal (CTM) [1] 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 non-standard-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 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].

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. 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 invent 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) supports an *application binary interface* (ABI) based the AMD CTM r5xx GPU ISA [1], and (ii) 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.

Note that it is NOT a goal of this work to compete with GPUs on regular floating point computation! Instead we envision that several aspects of our GPU-inspired architecture can be extended in future implementations to better capitalize on the strengths of FPGAs: rather than focusing on floating-point computation, we can instead focus on non-standard bit-width computation or other custom functions; we can also scale the soft processor in the vector dimension as in previous work [8, 9]; finally, we can scale the number of soft processor accelerators via multiprocessor implementations to fully-utilize available memory bandwidth.

1.3. Contributions

This paper makes the following contributions: (i) we propose a new GPU-inspired architecture and programming model for FPGA-based acceleration based on soft-processors that exploit multithreading, vector instructions and multiple processors; (ii) we 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) we demonstrate that these features, when implemented in concert, result in a soft processor design that can fully-utilize a deeply-pipelined datapath.

2. 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 AMD CTM SDK's application binary interface (ABI) that our soft processor implements.

2.1. 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 vector parallelism. 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

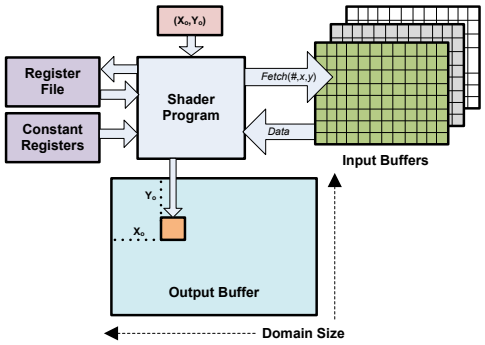


Fig. 1. The interaction of a shader program with memories and registers.

```
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
```

Fig. 2. CTM code for an example shader program for element-wise matrix multiplication plus an offset.

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.

2.2. 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 [1, 10], hiding many graphics-specific aspects of the GPU. A programmer can potentially create a shader program directly in the AMD CTM r5xx ISA, or else use a high-level language such as NVIDIA's `Cg` [11] or CUDA [12] which can then be compiled down to the r5xx ISA.[1] 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 *compute* 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.

Figure 2 shows CTM code for an example shader program for element-wise matrix multiplication plus an offset. 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; and (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 set-up by the CTM driver, but are otherwise read-only. Note that each instance of a shader program has a context of general-purpose registers while it is executing.

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 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 (ie., s0). For example, the address for the sources given as `r0.rg s1` 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 upper-case, followed (on a new line) by a scalar operation specified in lower case. In the example the ALU instruction is a pair of *multiply-add*s 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.

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.

3. 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.

[1] In this work we developed a software flow from `Cg` to r5xx via the `cgc` compiler included in NVIDIA's `Cg` toolkit and Microsoft's pixel shader virtual assembly language (`ps3`)—a similar flow could be developed for CUDA.

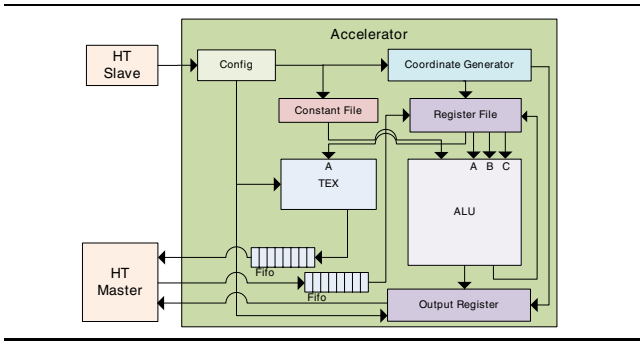


Fig. 3. Overview of a GPU-like accelerator, connected to a Hypertransport (HT) master and slave.

3.1. Overview

Figure 3 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.

Coordinate Generation As described in Section 2, 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 2.2) 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.

Output Similar to input buffers, the base addresses and widths of the output buffers are pre-configured 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).

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	...	...	...

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.

3.2. Tolerating Limited Memory Ports

In Figure 3 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 read 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 attempt 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.

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.

3.3. 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 non-ideal sequences of instructions will be undesirable bubbles in the two datapaths. The second reason 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. 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 issue 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.

3.4. Implementing the 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 8KB of on-chip memory per batch. For example, as illustrated in Figure 4, 32 batches would require 256KB of on-chip memory, which means that we must use four of the 64KB M-RAM blocks available in the Stratix II chip in the XD1000 module. Figure 5 shows the circuit we use to transpose the operands read across threads in a batch for ALU instructions so that

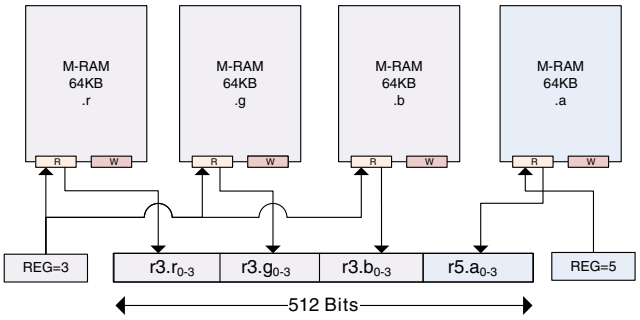


Fig. 4. Mapping our register file architecture to four Stratix II's 64KB M-RAM blocks. The read circuitry shows an example where we are reading operands across threads in a batch for a vector/scalar ALU instruction pair (VLIW): `r3` as an operand for the vector instruction and `r5` as an operand for the scalar instruction. While not shown, register writes are implemented similarly.

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.

4. MEASUREMENT METHODOLOGY

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 100MHz. We choose this frequency to match the 100MHz HT IP core, which in turn is designed to match a 4x division of the physical link clock frequency (that is 400MHz). 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 220MHz, 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

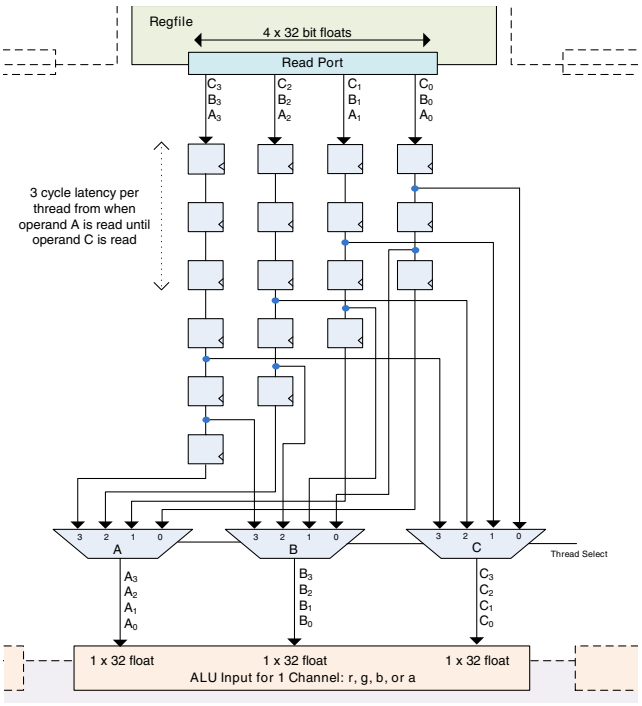


Fig. 5. A circuit for transposing the thread-interleaved operands read from the central register file into a correctly-ordered sequence of operands for the ALU datapath.

the latency of an HT read request as a sum of the individual latencies of the sub-components involved. We assume that our soft processor is running at 100MHz 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 51ns; 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.* [13], and the latencies for the the host HT controller, DDR controller, and DDR access were obtained from Holden [14]. 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. In our initial experiments configured with an 8-bit HT link, memory was indeed a bottleneck limiting ALU utilization. Hence in our evaluation here we investigate the impact of a 16-bit HT link such as the one described in [13]. Our HyperTransport model is somewhat idealized since we do not account for possible HT errors nor contention by the host CPU for memory.

Cycle-Accurate Simulation Our simulator is cycle-accurate

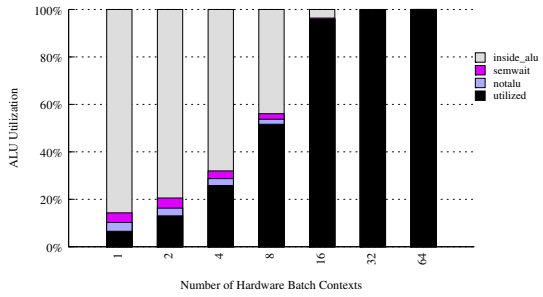
at the block interfaces shown in Figure 3. For each block we estimate a latency based on the operations and data-types present in a behavioral C code implementation. We assume that the batch issue logic is fully pipelined,[2] allowing us to potentially sustain the instruction issue schedule shown previously in Table 1.

Benchmarks Since our system is compatible with the interface specified by CTM we can execute existing CTM applications, including Cg applications, by simply re-linking the CTM driver to our simulation infrastructure. We evaluate our system using the following three applications that have a variety of behavior and ratios of ALU to TEX instructions (shown in parentheses): MATMATMULT (2.25) which performs dense matrix-matrix multiplication, SGEMM (2.56) which is a core routine of the BLAS math library, and PHOTON (4.00) which is a kernel from a Monte Carlo radiative heat transfer simulation.

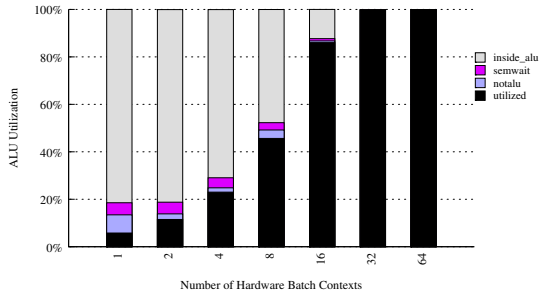
5. UTILIZATION AND PERFORMANCE

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 provides a greater opportunity for fully-utilizing the pipeline and avoiding bubbles by scheduling instructions to issue across a larger number of threads.

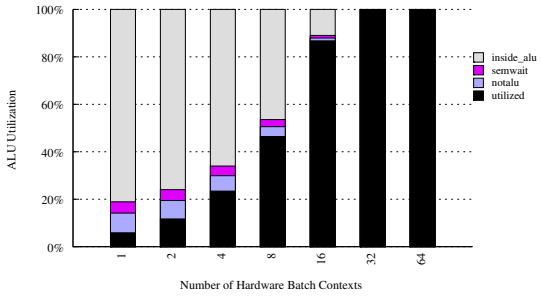
Figure 6 shows ALU utilization for a varying number of hardware batch contexts—from one to 64 batches. Since each batch contains four threads, this means that we support from four to 256 threads. We limit the number of batch contexts to 64 because this design includes a central register file that consumes 512KB of on-chip memory, and thus eight of the nine M-RAMs available in a Stratix II FPGA (64KB 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: (i) **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); (ii) **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; (iii) **NotALU**: the next instruction is not an ALU instruction.



(a) PHOTON



(b) MATMATMULT



(c) SGEMM

Fig. 6. ALU datapath utilization.

From the figure we observe that when only one hardware batch context is supported that the ALU datapath is severely underutilized (less than 10% utilization), and that 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 32 batches, at which point all three applications achieve 100utilization; correspondingly, none of the three applications can benefit from further increasing to 64 batches. We note that neither waiting on memory semaphores (Semwait) nor lack of ALU instructions (NotALU) are significant components of idle cycles for any of the experiments. The fact that 32 batches seems sufficient makes intuitive sense since 32 batches comprises 128 threads, while the ALU datapath pipeline

[2]More details on our batch issue design are available in Kingyens' MASc thesis [15]

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.

5.1. 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 3.4 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 this batch state 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 M4K memory blocks.

6. 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 and vector operations 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 fully-exploit the potential of FPGA-based acceleration platforms; we believe that GPU-inspired programming models and architectures are a step in the right direction.

7. REFERENCES

[1] J. Hensley, "Amd ctm overview," in *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, 2007.

[2] D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with direct memory access from ansi/iso standard c functions," *FCCM*, April 2006.

[3] J. L. Tripp, K. D. Peterson, C. Ahrens, J. D. Poznanovic, and M. Gokhale, "Trident: An fpga compiler framework for floating-point algorithms." *FPL*, 2005.

[4] J. Koo, D. Fernandez, A. Haddad, and W. Gross, "Evaluation of a high-level-language methodology for high-performance reconfigurable computers," *ASAP*, July 2007.

[5] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for sopc area reduction," *FCCM*, April 2006.

[6] M. Labrecque and J. Steffan, "Improving pipelined soft processors with multithreading," *FPL*, 2007.

[7] R. Moussali, N. Ghanem, and M. A. R. Saghir, "Supporting multithreading in configurable soft processor cores," in *CASES*, 2007.

[8] P. Yiannacouras, J. G. Steffan, and J. Rose, "Vespa: Portable, scalable, and flexible fpga-based vector processors," in *CASES*, 2008.

[9] J. Yu, G. Lemieux, and C. Eagleston, "Vector processing as a soft-core cpu accelerator," in *FPGA*, 2008.

[10] M. Peercy, M. Segal, and D. Gerstmann, "A performance-oriented data parallel virtual machine for gpus," in *SIGGRAPH*, 2006.

[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 *SIGGRAPH*, 2003.

[12] D. Kirk, "Nvidia cuda software and gpu parallel computing architecture," in *ISMM*, 2007.

[13] D. Slogsnat, A. Giese, and U. Brüning, "A versatile, low latency hypertransport core," in *FPGA*, 2007, pp. 45–52.

[14] B. Holden, "Latency comparison between hypertransport and pci-express in communications systems, 2006." Online publication, 2006.

[15] J. Kingyens, "A GPU-Inspired Soft Processor for High-Throughput Acceleration," Master's thesis, University of Toronto, 2008.