A GPU-Inspired Soft Processor for High-Throughput Acceleration

by

Jeffrey Richard Code Kingyens

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

A GPU-Inspired Soft Processor for High-Throughput Acceleration

Jeffrey Richard Code Kingyens

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2008

In this thesis a soft processor programming model and architecture is proposed that is inspired by graphics processing units (GPUs) and well-matched to the strengths of FPGAs, namely highly-parallel and pipelinable computation. The proposed soft processor architecture exploits multithreading, vector operations, and predication to supply a floating-point pipeline of up to 60 stages via hardware support for up to 256 concurrent thread contexts. The key new contributions of this 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 is (i) programmable via NVIDIA's high-level `Cg` language, (ii) supports AMD's r5xx GPU ISA, and (iii) is realizable on an XtremeData XD1000 FPGA-based accelerator system, it is demonstrated that the proposed soft processor can achieve 100% utilization of the deeply-pipelined floating-point datapath.

Acknowledgements

It is a pleasure to acknowledge those who have helped me along the way. I am grateful for the much-valued advice and guidance provided by my advisor, Greg Steffan. I thank Angela Tran, for loving and being there for me each and every day. I truly appreciate the love and support from all of the Tran family. Finally, I thank my mom, dad and and sister who always have been proud of me. Their support has given me tremendous confidence over the years.

Contents

List of Tables

List of Figures

Chapter 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 [12, 14, 23], but currently have many limitations and often require the designer to massage their code to get the best result from synthesis.

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. While 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 new soft processor architecture that is more naturally capable of fully-utilizing the FPGA**.

1.1 A GPU-Inspired Programming Model and Architecture

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) [8] 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 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 [6, 13, 18]. In our high-throughput soft processor we essentially avoid stalls of any kind for very deeply pipelined functional units (up to 60 stages) via hardware support for many concurrent threads (currently up to 256 threads), and group threads into batches (also similar to a GPU) to decrease the overheads of managing threads individually.

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 [25, 26].

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 [16]. 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.

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 `SystemC` simulation of a GPU-inspired soft processor that (i) is programmable via NVIDIA's high-level `C`-based language called `Cg` [15], (ii) supports an *application binary interface* (ABI) based the AMD CTM r5xx

GPU ISA [8], 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. The long-term goal is to use this system to gain insight on how to best architect a soft processor and programming model for FPGA-based acceleration.

1.2 Research Goals

The focus of this dissertation is to develop and demonstrate the promise of a GPU-inspired soft processor architecture and programming model. To this end, we have the following goals:

1. To architect a GPU-inspired soft-processor that supports an existing GPU ISA and high-level programming language.

2. To overcome the port limitations of FPGA block memories in the design of the register file.

3. To avoid all bubbles for a deeply-pipelined floating-point datapath by tolerating memory and pipeline latency.

4. To build a simulation infrastructure to (i) estimate the performace of the proposed architecture on an XtremeData XD1000 FPGA-based acceleration system, and to (ii) evaluate the resulting utilization of the deeply-pipelined floating-point datapath.

1.3 Organization

This thesis is organized as follows. Chapter 2 highlights previous work related to the high-level programming of FPGA accelerators as well as the soft processor design optimization techniques we use in this work. Chapter 3 provides an overview of our programming model, which is primarily a background in the GPU hardware and programming model we are implementing. Chapter 4 defines the architecture of the hardware to be instantiated on the FPGA accelerator and highlights the design challenges and solutions. Chapter 5 describes how we are to map this architecture onto the Xtremedata XD1000 platform. Chapter 6 describes our simulation framework and the benchmark we have selected for experimentation. Chapter 7 presents the results of our utilization experiments. Chapter 8 draws together conclusions from our proposed programming model and results, and summarizes our overall contributions.

Chapter 2

Related Work

A common approach to compiling high-level code for FPGA-based acceleration is the use of behavioural synthesis techniques [4] to transform high-level input code, directly into hardware gates. In Section 2.1 we give an overview of these techniques and provide deeper insight into Trident, a synthesis-based hardware compiler for floating point C code in Section 2.1.1. Instead of generating a custom hardware circuit for computing a given task, our work uses soft processors to execute software code describing such a task. In Section 2.2 we give a general overview of soft processors and highlight some of the previous research that has explored soft processor design optimizations which we use to design our *high-throughput* soft processor.

2.1 Behavioural Synthesis-based Compilers

The main challenge of behavioural 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' ImpulseC, Altera's C2H [14], Trident [23], Mitrionics' Mitrion-C [12], SRC Computer's SRC Carte, ASC [17] and Celoxica's Handel-C [19]. Typically, these tools will compile C-like code to circuit descriptions in HDL which can then synthesized by standard FPGA CAD tools for deployment on accelerator systems such as the Xtremedata XD1000.

A synthesis-based compiler will exploit data-dependencies 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.

2.1.1 Trident

Trident [23] is the hardware synthesis tool most related to our work as it is primarily used for scientific algorithms requiring floating point types. Also, we are able to provide a more in depth treatment as the compiler is academic and open source, and therefore we are able to learn implementation details which are not available in commercial packages.

Trident's programming model is based on a subset of GNU C. Figure 2.1 shows a sample program that assumes 2 input matrices, named `A` and `B` of dimensions `WIDTH` and `HEIGHT` have been loaded into off-chip memory arrays. The scalar variables `x` and `y` defined as `extern` in the sample code refer to the primary inputs of the FPGA accelerator circuit. When new `x` and `y` values are written to this circuit the hardware accelerator performs a multiplication of the components of the two matrices and adds the offset 1.0. The result is stored in a third matrix, `C`. Declaring these matrix arrays as `extern float` indicates to the compiler that data is located in off-chip SRAM.

Trident off-chip memory is assumed to be very low-latency (1-3 cycles) SRAM directly connected to the FPGA. As such, the underlying scheduling algorithm of Trident depends on this fact for performance. There is no means of pre-fetching for the purpose of hiding latency. Long latency memory accesses will result in a low performance circuit. For deployment in a system like the Xtremedata XD1000, DMA engines are instantiated on the FPGA such that

```
extern float x, y;
extern float A[WIDTH][HEIGHT];
extern float B[WIDTH][HEIGHT];
extern float C[WIDTH][HEIGHT];

void multadd() {
    float offset = 1.0f;
    C[x][y] = A[x][y]*B[x][y] + offset;
}
```

Figure 2.1: An example program for element-wise matrix multiplication plus an offset, described in `Trident`.

memory can first be transferred from system SDRAM to private FPGA SRAM. The Trident computation executes and the SRAM buffer is read back to system memory.

Trident is not able to *pipeline* code such as that shown in the example. Instead, there will be some fixed latency until the computation finishes for a given `x`, `y`. Then, a new computation may begin by writing new input values to the primary inputs. Essentially, the compiler does not have built in knowledge that we wish to perform this `multadd` computation repetitively, in a data-parallel manner. While not implying data-parallel code makes the programming model much more flexible, for code that *could* map well to a data-parallel model, Trident does not make any efforts to take advantage of the afforded parallelism.

2.2 Soft Processors

Soft processors are microprocessors instantiated on a FPGA fabric. Two examples of industrial soft processors are the Altera NIOS [2] and the Xilinx Microblaze [1]. 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 [24] 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. The following subsections describe more recent work extending soft processors to better exploit parallelism. We also describe in detail work on soft processor register file design as it is a central issue for this dissertation.

2.2.1 Vector Soft Processors

Yu *et. al.* [27] and Yiannacouras *et. al.* [25] 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.

2.2.2 Multi-threaded Soft Processors

Yiannacouras *et. al.* [16] use multi-threading in soft processor designs and have shown that it can improve area efficiency dramatically. While their work focuses on augmenting a 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.

2.2.3 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.* [10] 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 in [10] such as instruction memory sharing between processors could be applied in a multi-processor design of our high-throughput soft processors.

2.2.4 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.

Jones *et. al.* [11] implement a register file using logic elements as opposed to built-in SRAMs. However, they show that using this technique results a register file with very high area consumption, low clock frequency and poor scalability to a large number of registers.

Saghir *et. al.* [21] use multiple dual-port memories to implement a banked register file, allowing the writeback of two instructions per cycle in cases where access conflicts do not occur. In a sense, this is similar to the solution we support in our work. While they must rely on the compiler to schedule register accesses within a program such that writes are conflict-free, the fact that we execute multiple threads in lock-step allows us to build conflict-free accesses to a 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.

Chapter 3

System Overview

In this section we give an overview of our system as well as 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 3.1 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

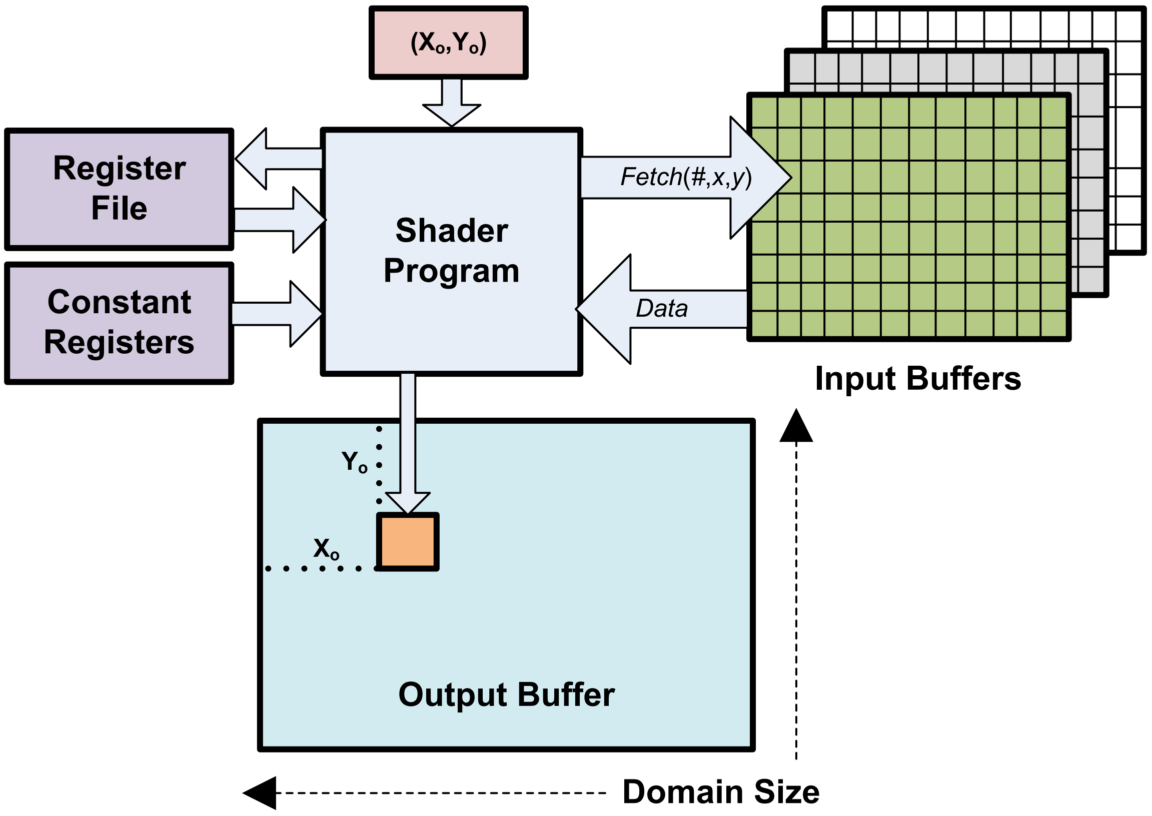


Figure 3.1: The interaction of a shader program with input and output memories. Input buffers can be randomly accessed while output is limited to a fixed location for each shader program instance—hence the execution of a shader program implies the invocation of parallel instances across all elements of the output buffer.

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` [15], a high-level, `C`-based programming language that targets GPUs. To give a taste of the `Cg` language, Figure 3.3 shows a sample program written in `Cg` for element-wise multiplication of two matrices, with the addition of a unit offset; for comparison, ANSI `C` code is provided for the same routine in Figure 3.2. In `Cg`, the `multadd`

```
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;
}
```

Figure 3.2: An example shader program for element-wise matrix multiplication plus an offset, described in `C`.

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 are in the `C` version. 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 3.1). 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

```
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;
}
```

Figure 3.3: An example shader program for element-wise matrix multiplication plus an offset, described in `Cg`.

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 [8, 20], hiding many graphics-specific aspects of the GPU. As illustrated in Figure 3.4, 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 *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 3.1). The host program also uses the CTM driver to load shader program binaries onto the GPU for execution.

Figure 3.5 shows the resulting CTM code from the example shader program in Figure 3.3. 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

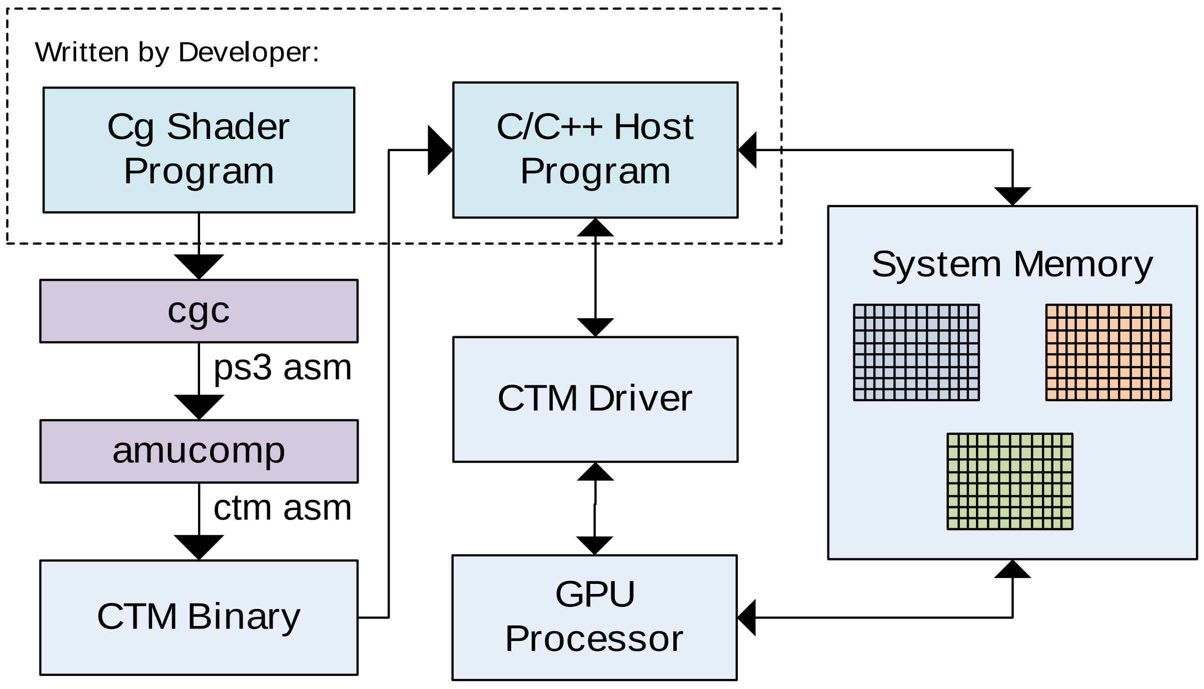


Figure 3.4: The software flow in our system. A software developer writes a high-level `Cg` shader program and a host program. The `Cg` shader program is translated into a CTM binary via the `cgc` and `amucomp` compilers and then folded into the host program.

```
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
```

Figure 3.5: An example shader program for element-wise matrix multiplication plus an offset, described in CTM assembly code.

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 3.1) 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.

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 (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 a VLIW operation-pair 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.

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 [20].

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.

Chapter 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 4.1 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 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

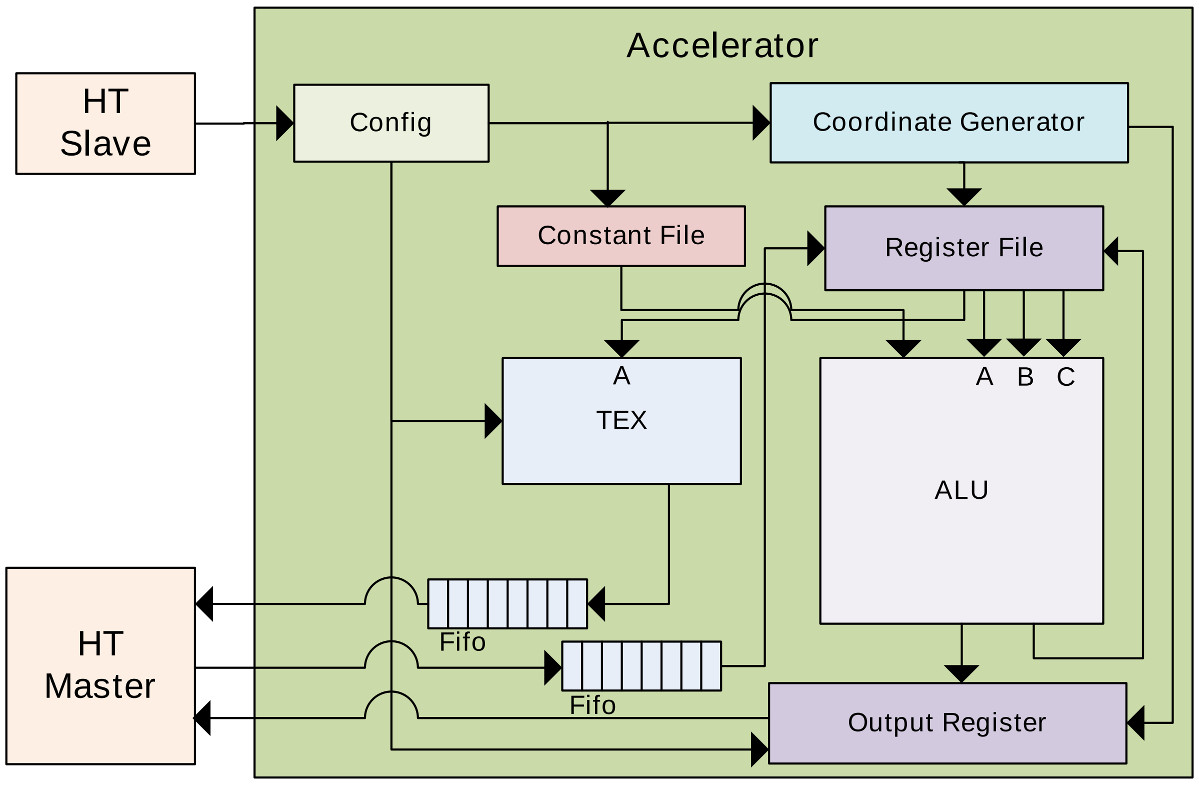


Figure 4.1: Overview of a GPU-like accelerator, connected to a Hypertransport (HT) master and slave.

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 to registers, 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 (eg., $>, <=, ==, !=$). 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 (FLW) that provide control of hardware-level call/return stacks, and branch and loop-depth hierarchies. As the FLW datapath does not interact with the register file it is not shown in Figure 4.1. We defer our explaination of control flow instructions to Section 4.3.1.

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 4.1 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

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 4.1: 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.

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, with this solution you cannot have 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 schedule 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 4.1 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 access.

This transposed register file design also eases the implementation of write ports. In fact, the schedule in Table 4.1 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.

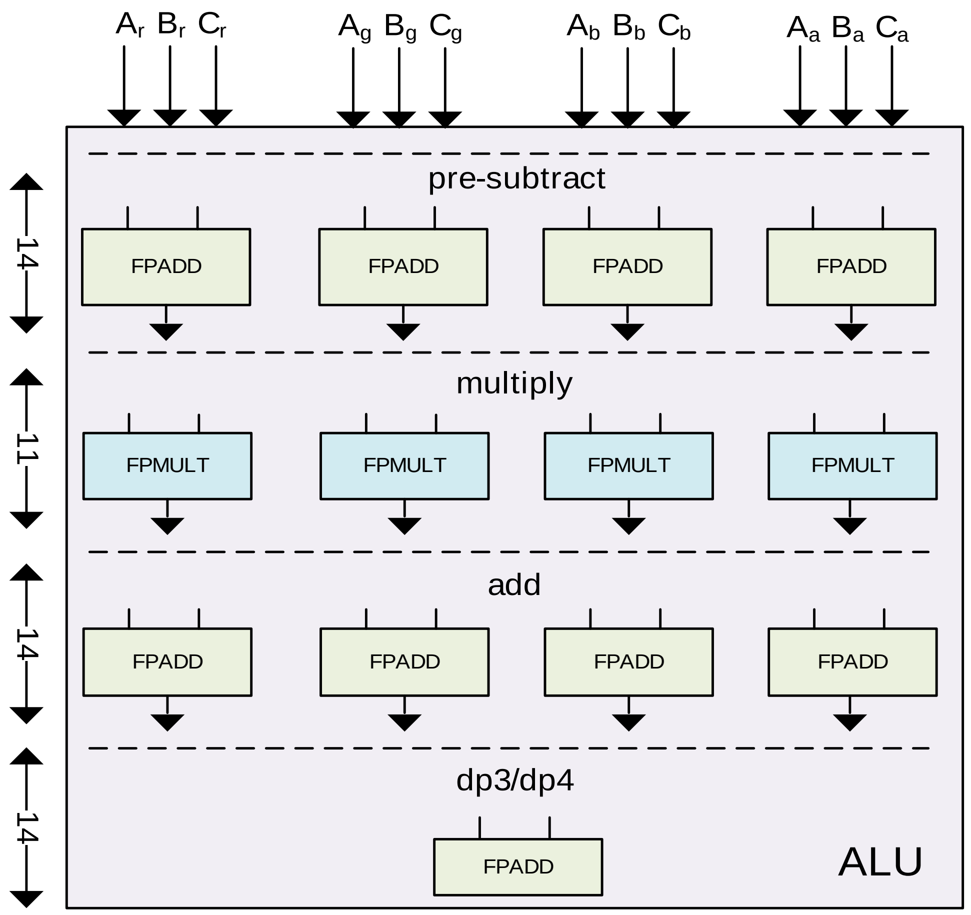


Figure 4.2: The floating point units in a datapath that supports MADD, DP3, and DP4 ALU instructions. The pipeline latency of each unit is shown on the left (for Altera floating point IP cores), and the total latency of the datapath is 53 cycles without accounting for extra pipeline stages for multiplexing between units.

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 4.1 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, as shown in Figure 4.2, 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 (more than 53 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 of threads in hardware, and dynamically switching between batches every cycle. We capitalize on the fact that all threads can be computed independently, 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 as illustrated in Table 4.2, 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.

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	...	...	...	...

Table 4.2: 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.

4.3.1 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 4.2 shows 2 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 FLW instruction 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.* [7] 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 [20].

Chapter 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 5.1, 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 multi-socket 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 400MHz DDR interface and can transfer 1.6GB/sec. The host CPU treats the XD1000 as an end-point 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 adding a

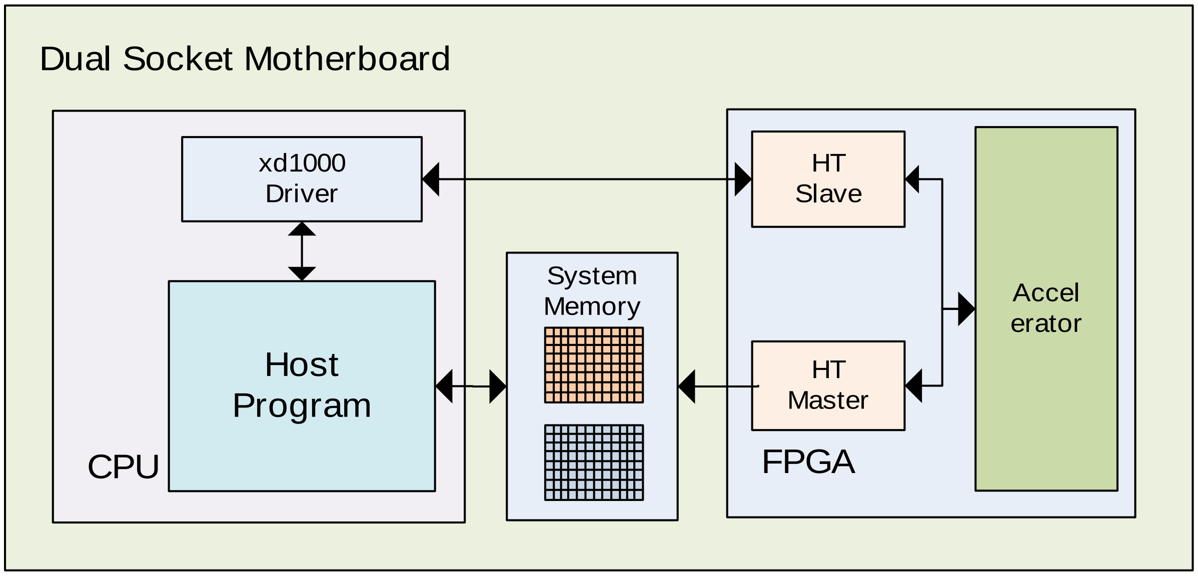


Figure 5.1: An XtremeData system with a XD1000 module.

driver layer on top of the XD1000 driver, and by memory-mapping the hardware configuration state registers and instruction memory of our accelerator 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.

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 memory capacity required, and (ii) performing the actual transposition. Each batch is composed of

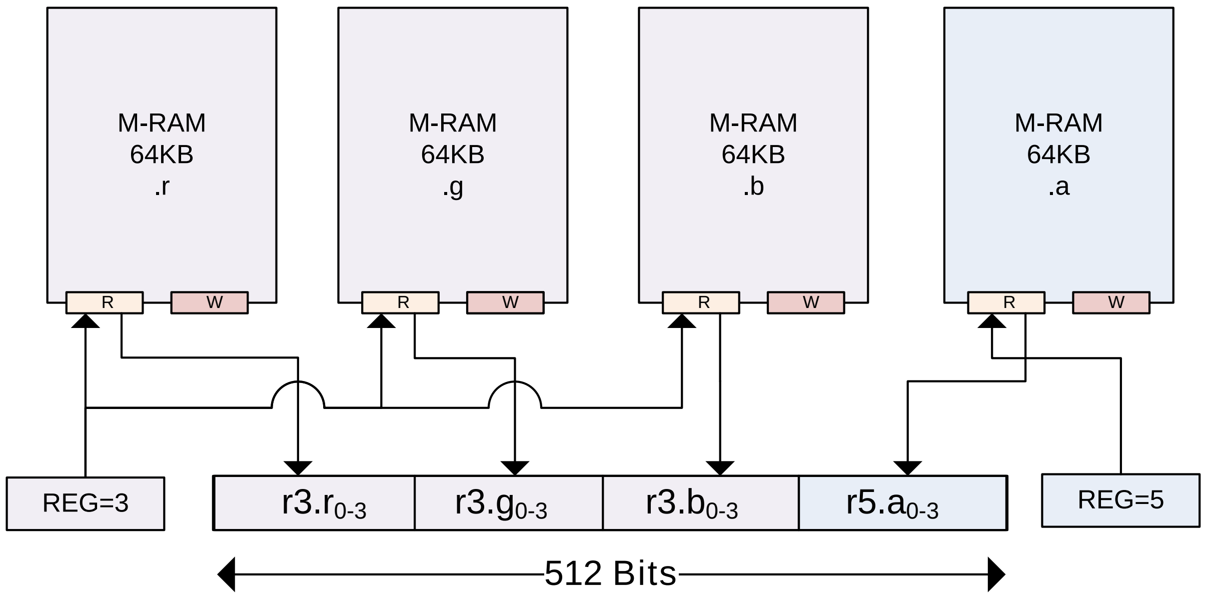


Figure 5.2: Mapping our register file architecture to four Stratix II's 64KB M-RAM blocks. The read circuitry shows an example where we are reading operands across threads in a batch for a vector/scalar ALU instruction pair (VLIW): `r3` as an operand for the vector instruction and `r5` as an operand for the scalar instruction. While not shown, register writes are implemented similarly.

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, 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, as illustrated in Figure 5.2. Figure 5.3 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 5.4 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 forward progress. The batch issue logic is pipelined, hence during a second cycle 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.

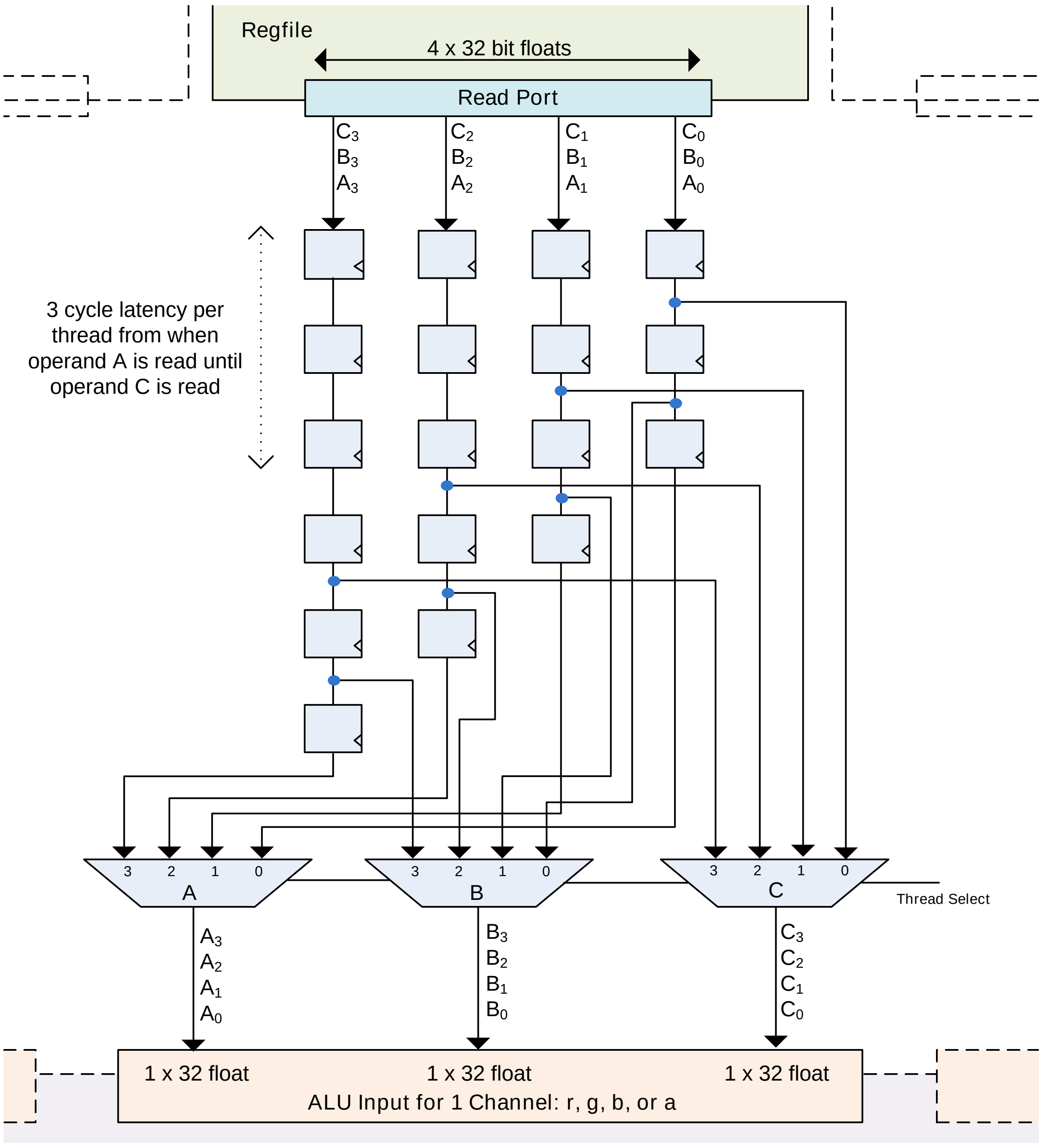


Figure 5.3: A circuit for transposing the thread-interleaved operands read from the central register file into a correctly-ordered sequence of operands for the ALU datapath.

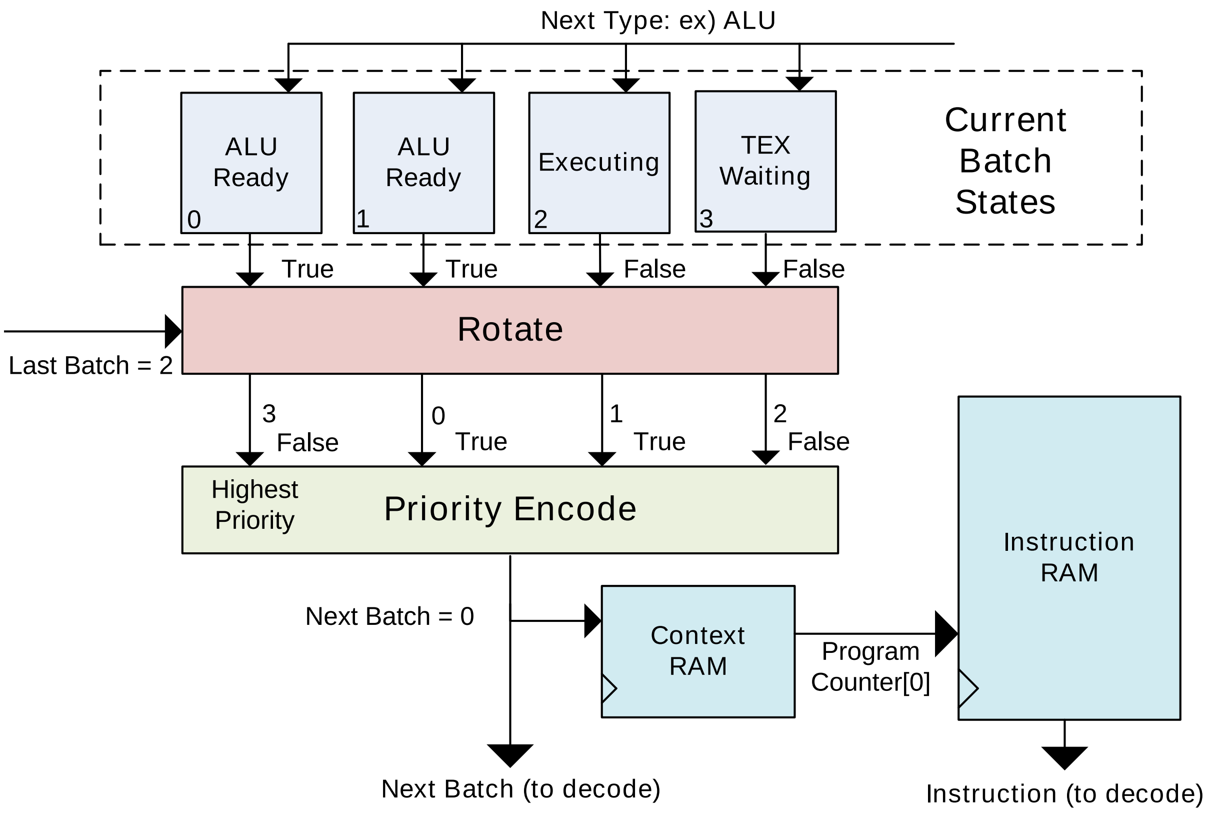


Figure 5.4: Batch issue logic for hardware managing 4 batch contexts.

Chapter 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 [3] to measure the ALU utilization and overall performance of workloads on our GPU-like soft processor.

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.

Cycle-Accurate Simulation Our simulator is cycle-accurate at the block interfaces shown in Figure 4.1. For each block we estimate a latency based on the operations and data-types present in a behavioral `C` code implementation. In most cases, we sketch the design of a circuit

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

Table 6.1: A breakdown of how each stage of an HT memory request contributes to overall access latency.

implementing the required C-code functionality and compute the latency of the critical path. We assume that the batch issue logic shown previously in Figure 5.4 is fully pipelined, allowing us to potentially sustain the instruction issue schedule shown previously in Table 4.2.

6.1.1 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.

Modeling of bandwidth requires cycle accurate communication between the output request FIFOs and the HT IP core. Each HT read packet can request up to 64 bytes of contiguous physical memory from system memory. As our threads are batched, each execution of a TEX instruction corresponds to 4 individual HT read packets sent out in sequence. Each HT read request is an 8 byte control packet identifying the end-chain device (FPGA) and the request address; in this case, from system memory. Data is returned in HT response packets consisting of a 4 byte control packet identifying the transaction ID, error status, and payload size, followed by a data packet containing the actual data. While it is highly likely that threads within a batch will request copies of some overlapping data due to *program locality*, we do not assume there is hardware available to merge these requests to reduce communication overhead. Hence, each *thread* generates one HT request which typically returns a vector register of data (16 bytes).

When a batch completes, write request packets are sent to the HT IP core. A write request consists of an 8 byte request control packet identifying the source device, length of the data to write and the target address. This is followed by a data packet of up to 64 contiguous bytes. A thread can write 16 bytes to each of 4 possible output buffers. Writing to different buffers results in separate HT request packets as the output buffers are not interleaved in memory such that writing data is contiguous between them. However, the lock-step completion of threads within a batch allows us to merge write data from 4 threads to a single output buffer, reducing overhead. The coordinate generator shown in Figure 4.1 allocates the coordinates into batches such that each batch represents 4 contiguous coordinates. Therefore, it is possible to compute the system memory address for the first coordinate in the batch, and write output data for all 4 threads in a minimal number of individual packets.

To model latency, we impose a delay in simulation time between when the HT request is sent out and the response is available to the accelerator. This delay is constant and computed as a sum of individual latencies listed in Table 6.1. 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.* [22], and the latencies for the the host HT controller, DDR controller, and DDR access were obtained from Holden [9]. Our HyperTransport model is somewhat idealized since we do not account for possible HT errors nor contention by the host CPU for memory.

The HT protocol specification states that a device may only have up to 32 outstanding request packets. We model this by checking a counter and holding additional requests back in a queue if the outstanding count reaches 32.

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 re-linking the CTM driver to our simulation infrastructure. We evaluate our system using the following three applications that have a variety of instruction mixes and behavior. 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.* [5]. 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 $\mathbf{C_{new}} = \alpha(\mathbf{A} \cdot \mathbf{B}) + \beta\mathbf{C_{old}}$ 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 [23] 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.

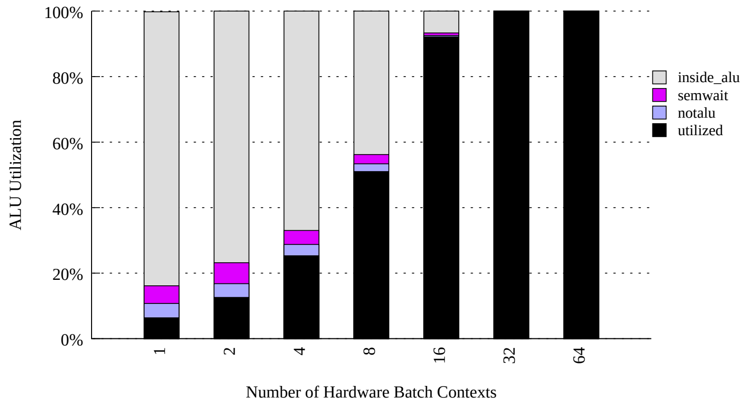
Chapter 7

Utilization and Performance

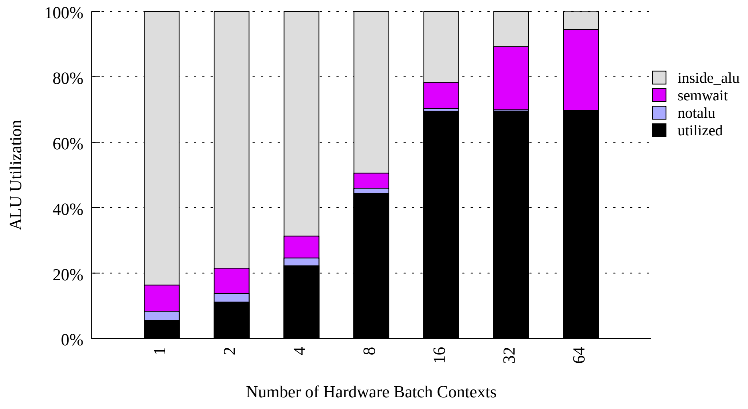
Our foremost goal is to fully-utilize the ALU datapath. In this section we measure the ALU datapath utilization for several configurations of our architecture, and also measure the impact on performance of increasing the number of hardware batch contexts. Recall that an increasing number of batches provides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

Figure 7.1 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 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.

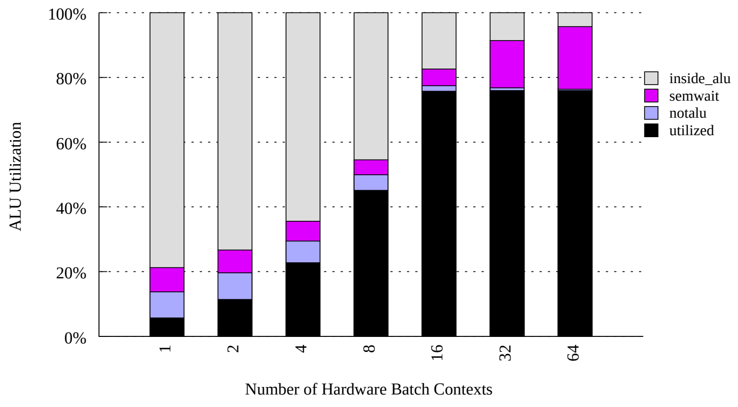
- *Semwait*: The next instruction is an ALU instruction, but it is waiting for a memory semaphore because it depends on an already in-flight TEX instruction (memory load).



(a) PHOTON



(b) MATMATMULT



(c) SGEMM

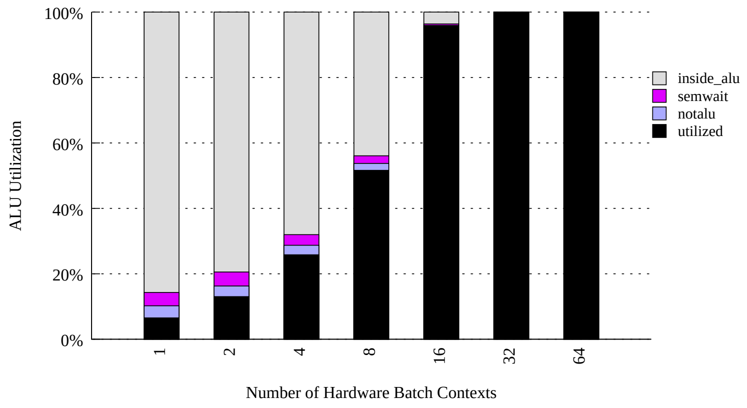
Figure 7.1: ALU datapath utilization for the 8-bit HT interface provided with the XD1000 system.

- *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 a previous ALU instruction completes before issuing a new one, to ensure that any register dependences are satisfied.

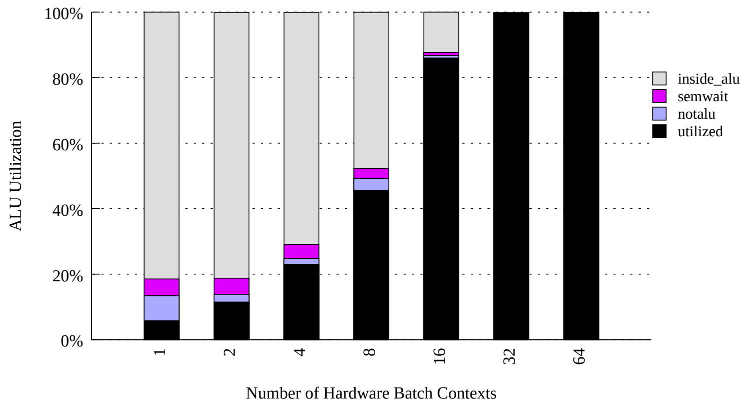
- *NotALU*: The next instruction is not an ALU instruction.

From the figure we observe that when only one hardware batch context is supported 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 (*executing*). 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).

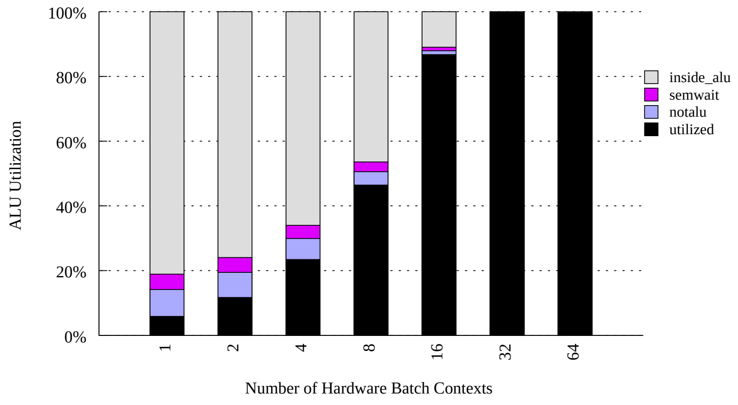
While the HT IP core provided for the XD1000 is limited to an 8-bit HT interface, the actual physical link connecting the FPGA and CPU is 16 bits. Since memory appears to be a bottleneck limiting ALU utilization, we investigate the impact of a 16-bit HT link such as the one described in [22] as shown in Figure 7.2. 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



(a) PHOTON



(b) MATMATMULT



(c) SGEMM

Figure 7.2: ALU datapath utilization for a 16-bit HT interface.

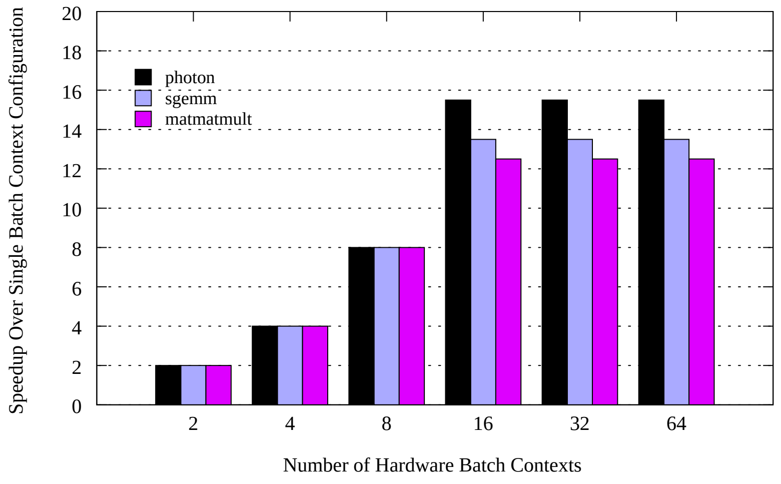
the ALU datapath pipeline is roughly only 53 cycles deep and thus requires only that many ALU instructions to be fully utilized—more deeply pipelined ALU functional units would likely continue to benefit from increased contexts.

While maximizing ALU datapath utilization is our overall goal, it is also important to understand the impact of increasing the number of hardware batch contexts on performance. Figure 7.3 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 sub-linear. For the 8-bit HT interface, performance does not improve beyond 16 contexts, while for the 16-bit HT interface 32 contexts provides an improvement but 64 contexts does not. Looking at the 16-bit HT interface for 32 contexts, we see that for 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. In detail, PHOTON benefits the least and has a ratio of 4.00, followed by SGEMM that has a ratio of 2.56, and MATMATMULT benefits the most and has a ratio of 2.25.

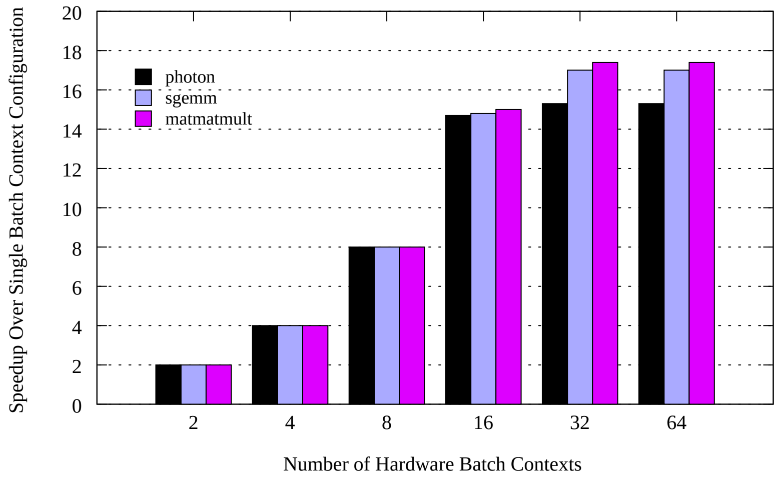
7.1 Reducing the Register File

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 a problem when instantiating a single soft processor, when trying to scale the number of processors for higher throughput, M-RAMs would be the limiting resource.

This base register file design supports the full 128 general purpose vector registers per thread defined by CTM, most of which 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,



(a) 8-bit HT interface.



(b) 16-bit HT interface.

Figure 7.3: Speedup vs a single hardware batch context for (a) 8-bit and (b) 16-bit HT interfaces.

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.

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.

Chapter 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 53-stage ALU datapath when 32 batch contexts are supported in hardware.

8.1 Contributions

This thesis (i) proposes 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) describes mechanisms for managing threads and register files that maximize data-level and instruction-level parallelism while overcoming the challenge of port limitations of FPGA block memories; (iii) demonstrates that these features, when implemented in concert, result in a soft processor design that can fully-utilize a deeply-pipelined datapath; (iv) contributes an expandable software simulator for executing the CTM programming specification, with performance estimates. This simulator will be used in future research and for reference in a hardware implementation.

8.2 Future Work

This work motivates several avenues of further research. While our current design supports floating-point based ALUs to remain compatible with the CTM interface, FPGAs would likely excel at other forms of computation such as fixed-point or non-standard-bit-width computation. It would also be beneficial to exploit much wider vector operations rather than the 4-element vectors defined by CTM. Beyond reducing the register file to match the needs of the application, there are many other avenues for customizing the architecture. Finally, the long term goal of this research is to discover new high-level programming models 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.

Appendix A

Cg Code

A.1 Photon

```
struct data_in {
    float3 color : COLOR0;
    float2 coord : TEXCOORD0;
};

struct data_out {

    float2 data : COLOR;
};

data_out main(data_in IN, uniform sampler2D delrhs : TEXUNIT0,
                          uniform sampler2D pos:   TEXUNIT1,
                          uniform sampler2D sqle : TEXUNIT2,
                          uniform sampler2D ls :   TEXUNIT3)
{
data_out OUT;
float ssq = 0.0;
OUT.data.x = 0.0;

// fetch the l, le and li value
float3 tls = tex2D(ls, IN.coord).xyz;
if(tls.x != tls.y) {  // l != le

float4 sqlet = tex2D(sqle, IN.coord).xyzw;
float4 delrhst = tex2D(delrhs, IN.coord).xyzw;
float det = dot(sqlet.yz, delrhst.yx);  // det = ex*delyl - ey*delxl;
float absdet = abs(det);
if(absdet <= 1.0000000133514319600181e-10)
    det = 1.0000000133514319600181e-10;
float dtinv = 1.0/det;
float xi = dtinv * (delrhst.x*delrhst.z - sqlet.y*delrhst.w);
float yi = dtinv * (delrhst.y*delrhst.z - sqlet.z*delrhst.w);
float4 post = tex2D(pos, IN.coord).xyzw; // fetch position information
ssq = (xi - post.x)*(xi - post.x) + (xi - post.z)*(xi - post.z) +
      (yi - post.y)*(yi - post.y) + (yi - post.w)*(yi - post.w);
if(ssq <= sqlet.x) {
OUT.data.x = tls.x;
}
}

OUT.data.y = ssq;
return OUT;
}
```

A.2 Sgemm

```
sampler2D mAX   : register(s0);
sampler2D mAY   : register(s2);
sampler2D mAZ   : register(s4);
sampler2D mAW   : register(s6);

sampler2D mBX   : register(s1);
sampler2D mBY   : register(s3);
sampler2D mBZ   : register(s5);
sampler2D mBW   : register(s7);

float4  step    : register(c0);
int     N       : register(i0);

struct block_t
{
    float4 X, Y, Z, W;
};

void multiply_block( inout block_t C, in float3 pos )
{
    float4 AX = tex2D( mAX, pos.yz ), BX = tex2D( mBX, pos.xy );
    float4 AY = tex2D( mAY, pos.yz ), BY = tex2D( mBY, pos.xy );
    float4 AZ = tex2D( mAZ, pos.yz ), BZ = tex2D( mBZ, pos.xy );
    float4 AW = tex2D( mAW, pos.yz ), BW = tex2D( mBW, pos.xy );

    C.X.xyzw += AX.xxzz*BX.yzyz + AX.wwyy*BX.wxwx + AY.xxzz*BZ.yzyz +
                AY.wwyy*BZ.wxwx;
    C.Y.xyzw += AX.xxzz*BY.yzyz + AX.wwyy*BY.wxwx + AY.xxzz*BW.yzyz +
                AY.wwyy*BW.wxwx;
    C.Z.xyzw += AZ.xxzz*BX.yzyz + AZ.wwyy*BX.wxwx + AW.xxzz*BZ.yzyz +
                AW.wwyy*BZ.wxwx;
    C.W.xyzw += AZ.xxzz*BY.yzyz + AZ.wwyy*BY.wxwx + AW.xxzz*BW.yzyz +
                AW.wwyy*BW.wxwx;
}

block_t main(float2 vpos : VPOS) : COLOR
{
    float3 pos = vpos.xyy*step.ywy;
    block_t C = {0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0};
    for (int i = 0; i < N.x; i++)
    {
        multiply_block( C, pos );    pos.y += step.y;
        multiply_block( C, pos );    pos.y += step.y;
        multiply_block( C, pos );    pos.y += step.y;
        multiply_block( C, pos );    pos.y += step.y;
    }

    return C;
}
```

Appendix B

CTM Code

B.1 Photon

```
I000:   /u/i TEX r1 r0.rgrr s0
I001:   /u/i TEX r2.ooox r0.rgrr s2
I002:   /u/i TEX r3.ooxx r0.rgrr s3
I003:   /p/u/i/v TEX r0 r0.rgrr s1
I004:   /p MAD (r1 r2 r3) r4.oox src0.gr0 src1.gb0 src0.000
mad (c0 c0 c0) r2 src2.r src2.1 -src2.g
I005:   MAD (r4 r2 c0) r2.xox src0.rar src1.0g0 src0.000
mad (r1 c0 c0) r3 src0.r src0.1 src0.g
I006:   MAD (r2 c0 c0) r2.xxo src1.rra src0.00b src0.000
cmp (r3 r1 c0) r1 src0.1 src0.0 -|src0|
I007:   MAD (r1 r2 c0) r1.oxx src0.r00 src0.b00 -src1.g00
cmp (r2 r1 c0) r1 src0.0 src1 -|src0|
I008:   MAD (r1 r2 c0) r2.xox src0.0g0 src0.0b0 -src1.0b0
cmp (r1 r3 c0) r3 src1 src0.0 -src0
I009:   MAD r2.xxo r3.00r c0.111 -c0.00g
rcp r1 r3
I010:   MAD (r1 c0 c0) r1.oxo src0.ara src0.r0r src0.000
mad (r1 c0 c0) r0.x src0.0 src0.0 src0.0
I011:   MAD (r2 c0 c0) r2.xox src0.rar src0.0g0 src0.000
mad (r1 c0 c0) r0.x src0.0 src0.0 src0.0
I012:   D2A (sub r0 r1 r2) r0.oxx srcp.rb0 srcp.rb0 src0.000
mad (bias r0 c0 c0) r0 -src0 src2.1 src2.g
I013:   MAD (sub r0 r2 c0) r0.xxx src0.000 src0.000 src0.000
mad (bias c0 c0 c0) r3 srcp.g srcp.g src0.r
I014:   MAD (c0 c0 c0) r0.xxx src0.000 src0.000 src0.000
mad (r0 r3 c0) r3 src0 src0 src1
I015:   CMP (r2 c0 c0) r0.oxx src0.000 src0.arr -|src0.b00|
mad (r3 c0 c0) r0.x src0.0 src0.0 src0.0
I016:   MAD r0.xxx o0.xox r0.0r0 c0.111 c0.000
mad r0.x o0.x c0.0 c0.0 c0.0
I017:   MAD (sub r0 r2 c0) r0.xxx src0.000 src0.000 src0.000
cmp (bias c0 c0 c0) r3 src0.1 src0.0 srcp.r
```

```
I018:   MAD (c0 c0 c0) r0.xxx src0.000 src0.000 src0.000
cmp (r2 r3 c0) r3 src0.0 src1 -|src0|
I019:   CMP (r3 c0 c0) r0.xxx o0.oxx src0.000 src0.r00 -src0.arr
mad (r3 c0 c0) r0.x o0.x src0.0 src0.0 src0.0
END
HALT
```

B.2 Matrix Multiplication

```
MAD r1 r0.0g0 1.0 0.0
mad r1 0.0 1.0 0.0
MAD r2 r0.r00 1.0 0.0
mad r2 0.0 1.0 0.0

MAD r7 0.0 0.0 0.0
mad r7 0.0 0.0 0.0
MAD r8 0.0 0.0 0.0
mad r8 0.0 0.0 0.0
MAD r9 0.0 0.0 0.0
mad r9 0.0 0.0 0.0
MAD r10 0.0 0.0 0.0
mad r10 0.0 0.0 0.0

LOP done i0 jumpNoGo
start:

/w /u TEX r3 r1 s0
/u TEX r5 r2 s1
/u TEX r11 r1 s2
/p /v /u TEX r6 r2 s3

/p MAD r7 r3.rrb r5.rgr r7
mad r7 r3.b r5.g r7

/u TEX r4 r1 s4
/u TEX r13 r2 s5
/u TEX r12 r1 s6
/p /v /u TEX r14 r2 s7

MAD r7 r3.gga r5.bab r7
mad r7 r3.a r5.a r7

MAD r8 r3.rrb r6.rgr r8
mad r8 r3.b r6.g r8
MAD r8 r3.gga r6.bab r8
mad r8 r3.a r6.a r8

MAD r9 r4.rrb r5.rgr r9
mad r9 r4.b r5.g r9
MAD r9 r4.gga r5.bab r9
mad r9 r4.a r5.a r9

MAD r10 r4.rrb r6.rgr r10
```

```
mad r10 r4.b r6.g r10
MAD r10 r4.gga r6.bab r10
mad r10 r4.a r6.a r10

/p MAD r7 r11.rrb r13.rgr r7
mad r7 r11.b r13.g r7
MAD r7 r11.gga r13.bab r7
mad r7 r11.a r13.a r7

MAD r8 r11.rrb r14.rgr r8
mad r8 r11.b r14.g r8
MAD r8 r11.gga r14.bab r8
mad r8 r11.a r14.a r8

MAD r9 r12.rrb r13.rgr r9
mad r9 r12.b r13.g r9
MAD r9 r12.gga r13.bab r9
mad r9 r12.a r13.a r9

MAD r10 r12.rrb r14.rgr r10
mad r10 r12.b r14.g r10
MAD r10 r12.gga r14.bab r10
mad r10 r12.a r14.a r10

MAD r1.oxx 1.0 1.0 r1.r00
frc r0.x 0.0
MAD r2.xox 1.0 1.0 r2.0g0
frc r0.x 0.0

ELP start i0 jumpGo jumpIfAny

done:
MAD r0.xxx o0 r7 1.0 0.0
mad r0.x o0 r7 1.0 0.0
MAD r0.xxx o1 r8 1.0 0.0
mad r0.x o1 r8 1.0 0.0
MAD r0.xxx o2 r9 1.0 0.0
mad r0.x o2 r9 1.0 0.0
/p MAD r0.xxx o3 r10 1.0 0.0
mad r0.x o3 r10 1.0 0.0
HALT
```

B.3 Sgemm

```
main:
I000: MAD (c0 r0 c0) r17 src1.rgg src0.gag src0.000
 mad (c0 c0 c0) r18 src1.r src0.g src0.0
I001: MAD r18.xxo r17.00b c0.111 c0.000
 mad r14 c0.0 c0.1 c0.0
I002: MAD (r18 r0 c0) r19.xoo src0.0ba src0.111 src0.000
 mad (r18 c0 c0) r15 src1.g src1 src0.0
I003: MAD r14 c0.000 c0.111 c0.000
```

```
 mad r0.x c0.0 c0.0 c0.0
I004: MAD r13 r14 c0.111 c0.000
 mad r13 r14 c0.1 c0.0
I005: MAD r20 r14 c0.111 c0.000
 mad r16 r14 c0.1 c0.0
I006: MAD r16 r14 c0.111 c0.000
 mad r12 r14 c0.1 c0.0
I007: REP I118 b0 i0 jumpNoGo
I008: MAD r0.xxx c0.000 c0.000 c0.000
 mad r17 r15 c0.1 c0.0
I009: /w/u/i TEX r0 r17.abaa s0
I010: /u/i TEX r1 r17.rarr s1
I011: /u/i TEX r2 r17.abaa s2
I012: /u/i TEX r3 r17.rarr s3
I013: /u/i TEX r4 r17.abaa s4
I014: /u/i TEX r5 r17.rarr s5
I015: /u/i TEX r6 r17.abaa s6
I016: /p/u/i/v TEX r7 r17.rarr s7
I017: /p MAD (r0 r1 c0) r8 src0.aag src1.ara src0.000
 mad (r0 r1 c0) r8 src0.g src1.r src0.0
I018: MAD (r0 r1 r8) r8 src0.rrb src1.gbg src2
 mad (r8 c0 c0) r8 src0.b src1.b src0
I019: MAD (r2 r5 r8) r8 src0.rrb src1.gbg src2
 mad (r8 c0 c0) r8 src0.b src1.b src0
I020: MAD (r2 r5 r8) r8 src0.aag src1.ara src2
 mad (r2 r5 r8) r8 src0.g src1.r src2
I021: MAD (r0 r3 c0) r9 src0.aag src1.ara src0.000
 mad (r0 r3 c0) r0 src0.g src1.r src0.0
I022: MAD (r0 r3 r9) r0 src0.rrb src1.gbg src2
 mad (r0 c0 c0) r0 src0.b src1.b src0
I023: MAD (r2 r7 r0) r0 src0.rrb src1.gbg src2
 mad (r0 c0 c0) r0 src0.b src1.b src0
I024: MAD (r2 r7 r0) r0 src0.aag src1.ara src2
 mad (r2 r7 r0) r0 src0.g src1.r src2
I025: MAD (r1 r4 c0) r2 src0.ara src1.aag src0.000
 mad (r1 r4 c0) r1 src0.r src1.g src0.0
I026: MAD (r1 r4 r2) r1 src0.gbg src1.rrb src2
 mad (r1 c0 c0) r1 src0.b src1.b src0
I027: MAD (r5 r6 r1) r1 src0.gbg src1.rrb src2
 mad (r1 c0 c0) r1 src0.b src1.b src0
I028: MAD (r5 r6 r1) r1 src0.ara src1.aag src2
 mad (r5 r6 r1) r1 src0.r src1.g src2
I029: MAD (r3 r4 c0) r2 src0.ara src1.aag src0.000
 mad (r3 r4 c0) r2 src0.r src1.g src0.0
I030: MAD (r3 r4 r2) r2 src0.gbg src1.rrb src2
 mad (r2 c0 c0) r2 src0.b src1.b src0
I031: MAD (r6 r7 r2) r2 src0.rrb src1.gbg src2
 mad (r2 c0 c0) r2 src0.b src1.b src0
I032: MAD (r6 r7 r2) r2 src0.aag src1.ara src2
 mad (r6 r7 r2) r2 src0.g src1.r src2
I033: MAD (c0 c0 c0) r18.xox src0.rar src0.111 src0.0g0
 mad (r15 r12 r8) r8 src1 src2.1 src2
I034: /w/u/i TEX r3 r18.gbgg s0
I035: /u/i TEX r4 r18.agaa s1
I036: /u/i TEX r5 r18.gbgg s2
```

```
I037: /u/i TEX r6 r18.agaa s3
I038: /u/i TEX r7 r18.gbgg s4
I039: /u/i TEX r9 r18.agaa s5
I040: /u/i TEX r10 r18.gbgg s6
I041: /p/u/i/v TEX r11 r18.agaa s7
I042: /p MAD (r3 r4 c0) r12 src0.aag src1.ara src0.000
 mad (r3 r4 c0) r12 src0.g src1.r src0.0
I043: MAD (r3 r4 r12) r12 src0.rrb src1.gbg src2
 mad (r12 c0 c0) r12 src0.b src1.b src0
I044: MAD (r5 r9 r12) r12 src0.rrb src1.gbg src2
 mad (r12 c0 c0) r12 src0.b src1.b src0
I045: MAD (r5 r9 r12) r12 src0.aag src1.ara src2
 mad (r5 r9 r12) r12 src0.g src1.r src2
I046: MAD (add r8 r16 r12) r8 srcp src2.111 src2
 mad (bias r8 r12 c0) r8 src0 src1.1 src1
I047: MAD (r3 r6 c0) r12 src0.aag src1.ara src0.000
 mad (r3 r6 c0) r3 src0.g src1.r src0.0
I048: MAD (r3 r6 r12) r3 src0.rrb src1.gbg src2
 mad (r3 c0 c0) r3 src0.b src1.b src0
I049: MAD (r5 r11 r3) r3 src0.rrb src1.gbg src2
 mad (r3 c0 c0) r3 src0.b src1.b src0
I050: MAD (r5 r11 r3) r3 src0.aag src1.ara src2
 mad (r5 r11 r3) r3 src0.g src1.r src2
I051: MAD (add r0 r20 r3) r0 srcp src2.111 src2
 mad (add r0 r16 r3) r0 srcp src2.1 src2
I052: MAD (r4 r7 c0) r3 src0.ara src1.aag src0.000
 mad (r4 r7 c0) r3 src0.r src1.g src0.0
I053: MAD (r4 r7 r3) r3 src0.gbg src1.rrb src2
 mad (r3 c0 c0) r3 src0.b src1.b src0
I054: MAD (r9 r10 r3) r3 src0.gbg src1.rrb src2
 mad (r3 c0 c0) r3 src0.b src1.b src0
I055: MAD (r9 r10 r3) r3 src0.ara src1.aag src2
 mad (r9 r10 r3) r3 src0.r src1.g src2
I056: MAD (add r1 r13 r3) r1 srcp src2.111 src2
 mad (add r1 r13 r3) r1 srcp src2.1 src2
I057: MAD (r6 r7 c0) r3 src0.ara src1.aag src0.000
 mad (r6 r7 c0) r3 src0.r src1.g src0.0
I058: MAD (r6 r7 r3) r3 src0.gbg src1.rrb src2
 mad (r3 c0 c0) r3 src0.b src1.b src0
I059: MAD (r10 r11 r3) r3 src0.rrb src1.gbg src2
 mad (r3 c0 c0) r3 src0.b src1.b src0
I060: MAD (r10 r11 r3) r3 src0.aag src1.ara src2
 mad (r10 r11 r3) r3 src0.g src1.r src2
I061: MAD (add r2 r14 r3) r2 srcp src2.111 src2
 mad (add r2 r14 r3) r2 srcp src2.1 src2
I062: MAD (r18 c0 c0) r18.oox src0.g10 src0.1ar src1.gg0
 mad (r15 c0 c0) r0.x src0.0 src0.0 src0.0
I063: /w/u/i TEX r3 r18.rbrr s0
I064: /u/i TEX r4 r18.araa s1
I065: /u/i TEX r5 r18.rbrr s2
I066: /u/i TEX r6 r18.araa s3
I067: /u/i TEX r7 r18.rbrr s4
I068: /u/i TEX r9 r18.araa s5
I069: /u/i TEX r10 r18.rbrr s6
I070: /p/u/i/v TEX r11 r18.araa s7
```

```
I071: /p MAD (r3 r4 c0) r12 src0.aag src1.ara src0.000
 mad (r3 r4 c0) r12 src0.g src1.r src0.0
I072: MAD (r3 r4 r12) r12 src0.rrb src1.gbg src2
 mad (r12 c0 c0) r12 src0.b src1.b src0
I073: MAD (r5 r9 r12) r12 src0.rrb src1.gbg src2
 mad (r12 c0 c0) r12 src0.b src1.b src0
I074: MAD (r5 r9 r12) r12 src0.aag src1.ara src2
 mad (r5 r9 r12) r12 src0.g src1.r src2
I075: MAD (r3 r6 c0) r13 src0.aag src1.ara src0.000
 mad (r3 r6 c0) r3 src0.g src1.r src0.0
I076: MAD (r3 r6 r13) r3 src0.rrb src1.gbg src2
 mad (r3 c0 c0) r3 src0.b src1.b src0
I077: MAD (r5 r11 r3) r3 src0.rrb src1.gbg src2
 mad (r3 c0 c0) r3 src0.b src1.b src0
I078: MAD (r5 r11 r3) r3 src0.aag src1.ara src2
 mad (r5 r11 r3) r3 src0.g src1.r src2
I079: MAD (r4 r7 c0) r5 src0.ara src1.aag src0.000
 mad (r4 r7 c0) r4 src0.r src1.g src0.0
I080: MAD (r4 r7 r5) r4 src0.gbg src1.rrb src2
 mad (r4 c0 c0) r4 src0.b src1.b src0
I081: MAD (r9 r10 r4) r4 src0.gbg src1.rrb src2
 mad (r4 c0 c0) r4 src0.b src1.b src0
I082: MAD (r9 r10 r4) r4 src0.ara src1.aag src2
 mad (r9 r10 r4) r4 src0.r src1.g src2
I083: MAD (r6 r7 c0) r5 src0.ara src1.aag src0.000
 mad (r6 r7 c0) r5 src0.r src1.g src0.0
I084: MAD (r6 r7 r5) r5 src0.gbg src1.rrb src2
 mad (r5 c0 c0) r5 src0.b src1.b src0
I085: MAD (r10 r11 r5) r5 src0.rrb src1.gbg src2
 mad (r5 c0 c0) r5 src0.b src1.b src0
I086: MAD (r10 r11 r5) r5 src0.aag src1.ara src2
 mad (r10 r11 r5) r5 src0.g src1.r src2
I087: MAD (r18 c0 c0) r19.oxx src0.r00 src1.111 src1.g00
 mad (r8 r12 c0) r8 src0 src1.1 src1
I088: /w/u/i TEX r6 r19.rgrr s0
I089: /u/i TEX r7 r19.brbb s1
I090: /u/i TEX r9 r19.rgrr s2
I091: /u/i TEX r10 r19.brbb s3
I092: /u/i TEX r11 r19.rgrr s4
I093: /u/i TEX r13 r19.brbb s5
I094: /u/i TEX r14 r19.rgrr s6
I095: /p/u/i/v TEX r15 r19.brbb s7
I096: /p MAD (r6 r7 c0) r16 src0.aag src1.ara src0.000
 mad (r6 r7 c0) r12 src0.g src1.r src0.0
I097: MAD (r6 r7 r16) r16 src0.rrb src1.gbg src2
 mad (r12 c0 c0) r12 src0.b src1.b src0
I098: MAD (r9 r13 r16) r16 src0.rrb src1.gbg src2
 mad (r12 c0 c0) r12 src0.b src1.b src0
I099: MAD (r9 r13 r16) r16 src0.aag src1.ara src2
 mad (r9 r13 r12) r12 src0.g src1.r src2
I100: MAD (add r12 r8 r16) r16 srcp src2.111 src2
 mad (bias r8 r12 c0) r12 src0 src1.1 src1
I101: MAD (r6 r10 c0) r8 src0.aag src1.ara src0.000
 mad (r6 r10 c0) r6 src0.g src1.r src0.0
I102: MAD (r6 r10 r8) r6 src0.rrb src1.gbg src2
```

```
 mad (r6 c0 c0) r6 src0.b src1.b src0
I103: MAD (r9 r15 r6) r6 src0.rrb src1.gbg src2
 mad (r6 c0 c0) r6 src0.b src1.b src0
I104: MAD (r9 r15 r6) r6 src0.aag src1.ara src2
 mad (r9 r15 r6) r6 src0.g src1.r src2
I105: MAD (add r3 r0 r6) r20 srcp src2.111 src2
 mad (add r3 r0 r6) r16 srcp src2.1 src2
I106: MAD (r7 r11 c0) r0 src0.ara src1.aag src0.000
 mad (r7 r11 c0) r0 src0.r src1.g src0.0
I107: MAD (r7 r11 r0) r0 src0.gbg src1.rrb src2
 mad (r0 c0 c0) r0 src0.b src1.b src0
I108: MAD (r13 r14 r0) r0 src0.gbg src1.rrb src2
 mad (r0 c0 c0) r0 src0.b src1.b src0
I109: MAD (r13 r14 r0) r0 src0.ara src1.aag src2
 mad (r13 r14 r0) r0 src0.r src1.g src2
I110: MAD (add r4 r1 r0) r13 srcp src2.111 src2
 mad (add r4 r1 r0) r13 srcp src2.1 src2
I111: MAD (r10 r11 c0) r0 src0.ara src1.aag src0.000
 mad (r10 r11 c0) r0 src0.r src1.g src0.0
I112: MAD (r10 r11 r0) r0 src0.gbg src1.rrb src2
 mad (r0 c0 c0) r0 src0.b src1.b src0
I113: MAD (r14 r15 r0) r0 src0.rrb src1.gbg src2
 mad (r0 c0 c0) r0 src0.b src1.b src0
I114: MAD (r14 r15 r0) r0 src0.aag src1.ara src2
 mad (r14 r15 r0) r0 src0.g src1.r src2
I115: MAD (add r5 r2 r0) r14 srcp src2.111 src2
 mad (add r5 r2 r0) r14 srcp src2.1 src2
I116: MAD (r19 c0 c0) r0.xxx src0.000 src0.000 src0.000
 mad (c0 c0 c0) r15 src0.r src1.1 src1.g
I117: ERP I008 b0 i0 jumpGo jumpIfAny
I118: /p MAD r0.xxx o3 r14 c0.111 c0.000
 mad r0.x o3 r14 c0.1 c0.0
I119: MAD r0.xxx o2 r13 c0.111 c0.000
 mad r0.x o2 r13 c0.1 c0.0
I120: MAD r0.xxx o1 r20 c0.111 c0.000
 mad r0.x o1 r16 c0.1 c0.0
I121: MAD r0.xxx o0 r16 c0.111 c0.000
 mad r0.x o0 r12 c0.1 c0.0
 END
HALT
```

Bibliography

[1] Microblaze: http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm. World Wide Web electronic publication, 2008.

[2] Nios: http://www.altera.com/products/ip/processors/nios/nio-index.html. World Wide Web electronic publication, 2008.

[3] Guido Arnout. SystemC standard. In *ASP-DAC '00: Proceedings of the 2000 conference on Asia South Pacific design automation*, pages 573–578, New York, NY, USA, 2000. ACM.

[4] Raúl Camposano. Synthesis techniques for digital systems design. In *DAC '85: Proceedings of the 22nd ACM/IEEE conference on Design automation*, pages 475–481, New York, NY, USA, 1985. ACM.

[5] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, New York, NY, USA, 2004. ACM.

[6] Blair Fort, Davor Capalija, Zvonko G. Vranesic, and Stephen D. Brown. A Multithreaded Soft Processor for SoPC Area Reduction. *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 131–142, April 2006.

[7] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[8] Justin Hensley. AMD CTM overview. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 7, New York, NY, USA, 2007. ACM.

[9] Brian Holden. Latency Comparison Between HyperTransport and PCI-Express In Communications Systems, 2006. World Wide Web electronic publication, 2006.

[10] P. James-Roxby, P. Schumacher, and C. Ross. A single program multiple data parallel processing platform for FPGAs. *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 302–303, April 2004.

[11] A.K. Jones, R. Hoare, I.S. Kourtev, J. Fazekas, D. Kusic, J. Foster, S. Boddie, and A. Muaydh. A 64-way VLIW/SIMD FPGA architecture and design flow. *Electronics, Circuits and Systems, 2004. ICECS 2004. Proceedings of the 2004 11th IEEE International Conference on*, pages 499–502, Dec. 2004.

[12] J.J. Koo, D. Fernandez, A. Haddad, and W.J. Gross. Evaluation of a High-Level-Language Methodology for High-Performance Reconfigurable Computers. *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pages 30–35, July 2007.

[13] M. Labrecque and J.G. Steffan. Improving Pipelined Soft Processors with Multithreading. *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 210–215, Aug. 2007.

[14] David Lau, Orion Pritchard, and Philippe Molson. Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions. *Field-*

Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on, pages 45–56, April 2006.

[15] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM.

[16] Peter Yiannacouras Martin Labrecque and J. Gregory Steffan. Scaling Soft Processor Systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2008.

[17] O. Mencer. ASC: a stream compiler for computing with FPGAs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(9):1603–1617, Sept. 2006.

[18] Roger Moussali, Nabil Ghanem, and Mazen A. R. Saghir. Supporting Multithreading in Configurable Soft Processor Cores. In *CASES 2007*, pages 155–159, October.

[19] I. Page. Closing the gap between hardware and software: hardware-software cosynthesis at Oxford. *Hardware-Software Cosynthesis for Reconfigurable Systems (Digest No: 1996/036), IEE Colloquium on*, pages 2/1–211, Feb 1996.

[20] Mark Peercy, Mark Segal, and Derek Gerstmann. A performance-oriented data parallel virtual machine for GPUs. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 184, New York, NY, USA, 2006. ACM.

[21] Mazen A. R. Saghir, Mohamad El-Majzoub, and Patrick Akl. Datapath and ISA Customization for Soft VLIW Processors. *Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference on*, pages 1–10, Sept. 2006.

[22] David Slogsnat, Alexander Giese, and Ulrich Brüning. A versatile, low latency HyperTransport core. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international*

symposium on Field programmable gate arrays, pages 45–52, New York, NY, USA, 2007. ACM.

[23] Justin L. Tripp, Kristopher D. Peterson, Christine Ahrens, Jeffrey D. Poznanovic, and Maya Gokhale. Trident: An FPGA Compiler Framework for Floating-Point Algorithms. *FPL*, pages 317–322, 2005.

[24] Peter Yiannacouras, Jonathan Rose, and J. Gregory Steffan. The microarchitecture of FPGA-based soft processors. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 202–212, New York, NY, USA, 2005. ACM.

[25] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors. In *CASES'08: International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2008.

[26] Jason Yu, Guy Lemieux, and Christopher Eagleston. Vector processing as a soft-core CPU accelerator. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, 2008.

[27] Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector processing as a soft-core CPU accelerator. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 222–232, New York, NY, USA, 2008. ACM.