A Parameterizable SIMD Stream Processor

Asad Munshi `amunshi` Alexander Wong `a28wong` Andrew Clinton `ajclinto` Sherman Braganza `sbraganz` William Bishop `wdbishop` Michael McCool `mmccool`

`@uwaterloo.ca`

University of Waterloo
Waterloo, ON, Canada

Abstract

Stream Processing is a data processing paradigm in which long sequences of homogeneous data records are passed through one or more computational kernels to produce sequences of processed output data. Applications that fit this model include polygon rendering (computer graphics), matrix multiplication (scientific computing), 2D convolution (media processing), and data encryption (security). Computers that exploit stream computations process data faster than conventional microcomputers because they utilize a memory system and an execution model that increases on-chip bandwidth and delivers high throughput. We have designed a general-purpose, parameterizable, SIMD stream processor that operates on IEEE single-precision floating point data. The system is implemented in VHDL, and consists of a configurable FPU, execution unit array, and memory interface. The FPU supports pipelined operations for multiplication, addition, division, and square root. The data width is configurable. The execution array operates in lock-step with an instruction controller, which issues 32-bit instructions to the execution array. To exploit stream parallelism, the number of execution units as well as the number of interleaved threads is specified as a parameter at compilation time. The memory system allows all execution units to access one element of data from memory in every clock cycle. All memory accesses also pass through a routing network to support conditional reads and writes of stream data. Functional and timing simulations have been performed using a variety of benchmark programs. The system has also been synthesized into an Altera FPGA to verify resource utilization.

Keywords — *Stream processing, GPUs, SIMD.*

1 Introduction

Many applications of computers contain a high degree of inherent data parallelism. Such applications include graphics rendering, media processing, encryption, and image processing algorithms – each of which permit operations on different data elements to proceed in parallel. A fundamental problem of computer hardware design is to expose as much of this parallelism as possible without compromising the desired generality of the system. One such approach is stream processing, in which data parallelism is exposed by processing data elements independently and in parallel.

In stream processing, each record in a stream is operated on independently by a kernel – a small program applied to each data element, potentially in parallel. The result of applying a kernel to a stream produces another stream. The order in which streams and kernels are applied is very flexible, allowing operations to be optimized. Stream processing can be implemented in a way that minimizes hardware resource, allowing more computations to be performed using less area in a VLSI design and less memory bandwidth.

The research topic of stream processing has been pursued extensively at Stanford University as part of the development of the Imagine stream processor ([4], [3], [5], [1], [2]). The Imagine is a VLIW stream processor developed for media processing. The Imagine supports a 3-level memory hierarchy including off-chip RAM, a high-bandwidth stream register file, and high-speed local registers contained in each processing element. Researchers at Stanford have investigated the implementation of conditional streams [3], in which conditionals are implemented in a SIMD architecture through multiple passes and on-chip routing. Researchers have also investigated low-level (kernel) scheduling for VLIW instructions and high-level (stream control) scheduling [5] for placing and moving streams within the stream register file. Research into configurable parallel processors has been conducted in Texas [6] for the TRIPS processor.

Parallel processing has also undergone rapid development in the area of graphics processors (GPUs). Current-generation GPUs now support highly flexible instructions sets for manipulating pixel shading information, while exploiting fragment parallelism. Hardware features have even been proposed to make GPUs more like stream processors ([7], [8], [9]). Many researchers have recognized the potential performance exposed by GPUs and have ported a wide range of parallelizable algorithms to the GPU, including linear algebra operations, particle simulations, and ray tracing ([10], [11]). The introduction of general purpose GPU programming has led to the development of several high-level tools for harnessing the power of these devices. These attempts include the development of Brook for GPUs [12] (a project from Stanford, aimed at exposing the GPU as a general stream processor) and `Sh` ([14], [13], [15]) (a project at the University of Waterloo, which targets GPUs from ordinary host-based C++ code – an approach called metaprogramming.

This research project merges these two research efforts by developing xStream, a parameterizable stream processor that is compatible with existing APIs for GPU programming. The xStream permits most of the `Sh` system to be executed in hardware, including single-precision floating point operations. The system also supports a routing system that supports conditional streams. The system has been designed to be efficient, modular, easy to configure, and simple to program.

2 xStream Processor Overview

A high-level block diagram of the processor is shown in Figure 2. There are several key components of the design, including an array of floating point processing elements (PEs), the execution control unit, and an on-chip memory system consisting of an inter-unit routing network and fast stream register file. The DRAM and host are components that would be integrated eventually for a full platform design.

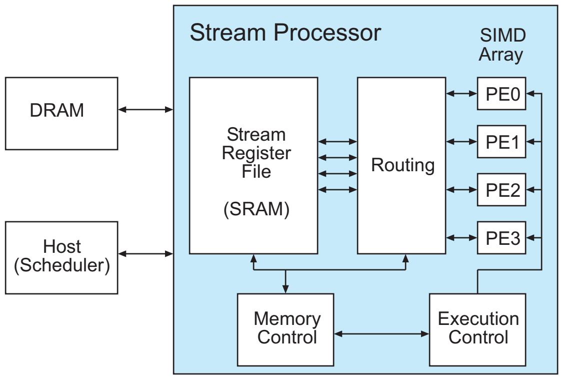

Figure 1: xStream Processor Block Diagram

The design implements the stream processing model by enforcing a restricted memory access model. All memory access from the execution unit array must pass through the stream register file, and all memory access must read or write at the head of the stream. The stream register file stores streams as a sequence of channels – each channel stores a component of a stream record. The channels are configured at startup of the device, and are dynamically read and written during execution.

Data parallelism is exploited in two major ways in the design. First, data elements can be processed in parallel by an array of processing elements. This results in a speedup of roughly N times for an array containing N processors, as there is little communication overhead for independent stream accesses. As the number of processors is increased, the memory and routing network automatically scale to match the new array size (with the routing only rising as $O(N * log_2 N)$. Second, interleaved multithreading is used within each processing element with an arbitrary number of threads, trading instruction level parallelism for the available (unlimited) data parallelism.

The execution model allows for conditional execution of read or write instructions to the memory bus. The system automatically routes stream data to the correct memory bank when processors are masked. Execution units are fully pipelined so (with multithreading) a throughput of 1 instruction per cycle is almost always achieved.

2.1 Floating Point Unit

The floating point arithmetic unit detailed in this document is based on a core parameterized floating point library developed at the Northeastern University Rapid Prototyping Lab (RPL) [16]. A number of extensions were made to the existing core library to provide additional arithmetic operations such as floating point division, square root, fraction and integer extraction operations. Furthermore, a parameterized floating point arithmetic unit interface was created to provide a single core component for general interfacing with the rest of the processor architecture.

2.1.1 Hardware Modules

The FPU core supports the following features:

- Implements a full parameterizable floating point structure with configurable mantissa and exponent bit lengths (single precision (32-bit) floating point format was used for the purpose of the project)
- Implements Floating point addition, subtraction (by negation), multiplication, division, square root, and fraction and integer extraction
- Implements two of the four IEEE standard rounding modes: round to nearest and round to zero
- Exceptions are partially implemented and reported according to the IEEE standard

The above specification was decided based on a number of important design decisions. For the purpose of the project, the parameterized floating point core unit was configured to comply with the IEEE 754 standard for single precision floating point representation. Therefore, the number of mantissa bits and exponent bits are set to 23 bits and 8 bits respectively, with a single sign bit. This provides the level of precision needed to perform basic floating point calculations, while still being manageable in terms of size and testing effort. Operations on denormalized floating point numbers are not supported because, based on research done by Professor Miriam Leeser at the Rapid Prototyping Lab [16], it would require extra hardware to implement and therefore is not necessary for our purposes.

The above list of floating point arithmetic operations was chosen as it provides the functionality needed for a subset of the `Sh` API and upon which all other operations can be built upon. Only two of the four IEEE standard rounding modes were implemented as the additional rounding modes ($+\infty$ and $-\infty$) were deemed unnecessary. Finally, only part of the exception handling specified by IEEE standard was implemented in the FPU core. The inexact exception is ignored while all other exceptions generate zero values at the output of the FPU along with a general exception flag. This was chosen to provide compatibility with the `Sh` API.

The FPU has been designed to operate using a six-stage pipeline. The first four stages involve the denormalizing of the floating point numbers and the arithmetic operation. The last two stages involve the rounding and normalizing of the floating point value.

2.2 Execution Control Unit

Several design goals have guided the development of the execution control unit in the xStream processor. These include:

- Compatibility with the `Sh` API
- Instruction set scalability and orthogonality
- Latency hiding for various levels of parallelism
- Interleaved multithreading support

The component diagram of an execution unit is shown in Figure 2.2. Note that each of the bus connections are multiplexed by the control signals arriving from the execution controller. The execution controller also includes a high-speed instruction cache for kernels.

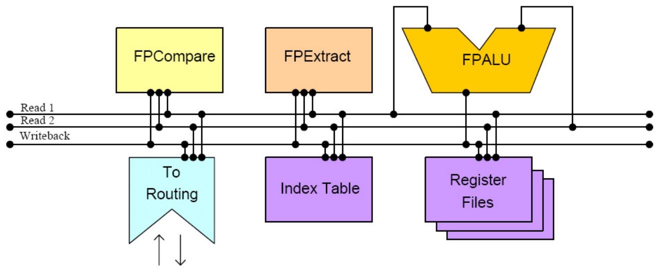

Figure 2: Execution Unit Component Diagram

2.2.1 Bus Structure

To allow a single instruction to complete every cycle requires sufficient register bandwidth to allow 2 input operands to be sent to the floating point unit and 1 output operand to be received from the floating point unit every cycle. This requires a 3-bus structure with 2 input buses and one output bus. One input bus and the output bus are also shared with the external routing interface for loading and storing external data to the register file.

2.2.2 Storage Structure

Each execution unit requires a local register file, which must be fast (single-cycle latency) and also must support a significant number of registers. Since a 3-bus structure was used, the register file uses 2 input and 1 output ports. A larger number of registers are required due to the limited access to memory in a stream architecture – so all constants, results, and temporaries must be stored in the register file.

In addition to local registers, many algorithms may require extended read-only storage for indexed constants or larger data structures. These can be stored in a block of read-only SRAM within each execution unit, into which indexed lookups may be performed. Data in this local memory must be transferred to registers before it can be operated on. These indexed memories can use a single input and output port.

2.2.3 Pipeline

The arithmetic units that are used within each execution unit include the components shown in Table 2.2.3. There are two different latencies for the units in the design, resulting in a complication when trying to pipeline the entire design. To create the pipeline for these units, we were faced with the choice of either fixing the pipeline length at the longer of the two latencies and incurring a performance penalty for the slower instructions, or supporting both latencies in separate pipelines and automatically resolving the structural hazards arising from this configuration. A design that allows for multiple variable pipeline lengths was chosen.

TABLE I
Functional Unit Latencies

Component	Latency
Floating point ALU	6 cycles
Floating point comparator	1 cycle
Indexed load/store	1 cycle

To automatically solve all hazard detection problems (structural and data hazards), we have elected to design a hardware writeback queue to track instructions in the pipeline.

1. When the desired register in the queue is already enabled – a conflict has been detected on the bus.
2. When any enabled queue register contains a writeback register equal to one of the registers being read – a data dependency needs to be avoided.

The hardware complexity to detect the first condition is trivial. For the second condition, a comparator is required for every queue slot for every read register. So in the case of 8 queue slots and 2 possible read registers, the number of 8-bit comparators required is 16.

2.2.4 Multithreading

The most important characteristic of the stream programming model is that each element can be operated on in parallel. To increase utilization of the pipeline and reduce the possibility of processor stalls, it is possible to take advantage of this characteristic by implementing multiple threads of execution within a single execution unit. There are several methods of implementing multithreading, ranging from very fine-grained (switching threads on every clock cycle) to coarse-grained (switching threads after a fixed time interval or only on costly operations like cache miss reads). Since the design does not permit any operation latencies of greater than 9 cycles, an efficient means of decreasing the effective latency is to support fine-grained interleaved multithreading. Interleaved multithreading is supported by duplicating the state-holding components in the design for each thread – thus, each register file needs to be duplicated for each thread. In addition, the control hardware needs to broadcast the currently active thread for each of the pipeline stages – decode, execute, and writeback.

2.2.5 Conditionals

Conditionals are handled in various ways in SIMD processors. In general, conditionals cannot be implemented directly in a SIMD processor because the processor array is required to remain in lockstep (ie. each processor executes the same instruction on a given clock cycle). Traditionally, conditionals have been implemented by masking the execution of processors for which a condition fails. In the xStream processor, conditionals are implemented using the method of predication, in which all processors first evaluate all conditional branches and then conditionally write back their results into the register file.

Conditionals are handled in the `Sh` intermediate representation through conditional assignment of 0.0 or 1.0 to a floating point value based on a comparison of two other floating point values. Although conditional support using floating point values is easy to implement, this approach leads to additional usage of floating point registers to store Boolean results. A more efficient alternative would be to support a new series of single-bit registers for storing Boolean values. Boolean operations were implemented using dedicated Boolean registers. This reduces the number of floating point instructions and relieves floating point registers from storing Boolean values.

2.2.6 Instruction Set

The instruction set includes all the operation shown in Table 2 with the given latency and functional unit.

TABLE II
Instruction Set

Instruction	Latency	Unit	Operation
NOP	4	BUS	No operation
ADD	7	FPU	Rd ← R1 + R2
MUL	7	FPU	Rd ← R1 ∗ R2
DIV	7	FPU	Rd ← R1 / R2
SQRT	7	FPU	Rd ← sqrt(R1)
INT	4	FRAC	Rd ← floor(R1)
FRAC	4	FRAC	Rd ← R1 − floor(R1)
CMP	4	CMP	Cd ← compare(R1, R2)
COND	4	BUS	Rd ← {R1 if C1}
LDI	4	IDX	Rd ← index(R1)
STI	4	IDX	index(R1) ← R2
GET	4	BUS	Rd ← {route(imm) if C1}
PUT	4	BUS	route(imm) ← {Rd if C1}
CON	4	BUS	Rd ← scater(imm)

Operation Notes:
1. `Rd` denotes a destination register. `R1` and `R2` denote the two register inputs.
2. `Cd` denotes a destination Boolean register. `C1` denotes an input Boolean register.
3. `imm` denotes an input immediate value.
4. `compare()` performs a comparison operation.
5. `index()` performs a lookup in the index constant table.
6. `route()` compacts outgoing data when execution units do not produce output and routes incoming data to the execution units requiring new elements.

Functional units are the components that are used during the execute (EX) stage of pipeline execution. The functional units that are available are shown in Table 2.2.6.

TABLE III
Functional Unit Descriptions

Label	Description
BUS	Internal execution unit buses
FPU	Single-precision floating point unit
FRAC	Integer/fractional part extraction unit
CMP	Single-precision floating point comparator unit
IDX	Index constant table

2.2.7 Instruction Controller

The active kernel is stored in SRAM local to the instruction control hardware. The control hardware is a simple state machine that reads and issues instructions. Because branching instructions are not supported and all data dependencies are resolved at compile time, the controller steps through the program and issues one instruction per cycle.

Instructions are decoded at the individual execution units, so the common instruction controller consists mainly of the state machine. The alternative is to decode instructions at the execution units, which would introduce a significant number of new signals to be routed from the instruction controller to the execution units. The xStream processor does not use this alternative approach.

2.3 Memory System

The on-chip memory system consists of the stream register file (SRF), SRF controller, and inter-element routing network. The memory system was designed with a number of design goals in mind. These include:
• Direct support for the stream and channel primitives exposed by `Sh`
• Bandwidth scalability to support arrays of execution units
• Hardware support for conditional stream operations

The stream register file is intended to store input, output, and temporary channels that are allocated during execution. A channel is a sequence of independent 32-bit floating point values. For operations on records containing many 32-bit values, the records must first be decomposed into channels and then stored into the SRF separately (for example, vectors ¡x,y,z¿ must be decomposed into an x-, y-, and z- channel before processing). The purpose of the stream register file is to provide high bandwidth on-chip storage component so that intermediate results can be stored and accessed quickly by the execution array. The

SRF is composed of several memory banks that permit parallel access. The number of banks in the SRF is coupled to the number of processors that need to make parallel access to the SRF – so for a processing array containing 8 processors, it is necessary to provide 8 equal-sized banks in the SRF memory.

2.3.1 Channel Descriptors

Only a single channel in the SRF can be accessed at once. When the channel is accessed, any number of processors can retrieve data from the channel. Since channels are always accessed in sequential order, it is necessary to keep track of the current address for access within that channel. This is done by storing an array of channel descriptors. Each channel descriptor consists of the current address and final address within that channel. When the current address reaches the final address, the channel is empty and any further references to that channel are rejected with an error. Figure 2.3.1 depicts how channels are striped across banks to allow for concurrent access.

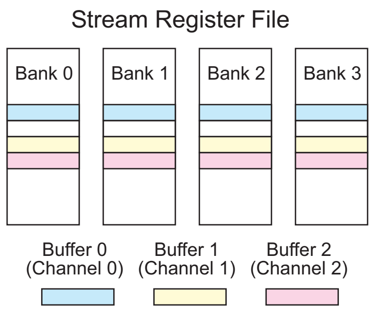

Figure 3: Description of Channels in SRAM

2.4 Routing Network

The routing network is what connects all the execution units to the SRAM banks. The same network is used to provide the processing elements with data and store the output back into RAM. It has been designed such that it can operate in either direction, from memory to PEs or from PEs to memory. The complexity involved with constructing such a network is based on three different aspects. Firstly, the network has to be parameterizable as it has to be able to expand to the number of PEs used during a simulation. Secondly, each PE has been designed with conditional read and conditional write commands. These conditions are based on the condition codes set during the execution of the program. Thus the network has to handle reading or writing a number of data elements that could be fewer than the most number of processing elements available. Lastly, the network has been constructed so that each PE does not need to worry about the next memory location from which data will be available. This is especially important given that each PE would not necessarily be reading data from memory whenever a read instruction is invoked. Thus, when a PE executes a read instruction, it shouldnt need to worry about this issue.

To accomplish the tasks mentioned above, the network was divided into two components, a barrel shifter and a compactor. The barrel shifter reorders the data being transferred. The compactor is what removes any gaps that may exist in the data. This is useful when certain PEs do not produce any data. This way data being stored to memory does not have any gaps between them. The same compactor can be used to introduce gaps between data elements when certain PEs do not need to read any new data. The compactor and the barrel shifter together provide data to each PE and write back any output produced while taking care of the three above mentioned issues.

3 Testing and Evaluation

Testing for the xStream processor has been carried out through functional simulations. There are three applications that we wrote to test the xStream processor. The three different programs show that the instruction set of the xStream processor has been designed to be accommodating for most tasks that can be broken up into parallel streams. The first is a fairly simple program that normalized a set of vectors. The second test was to generate a fractal image. The third test is a much larger project that is still in the works. It involves carrying out the process of rasterization of an image.

3.1 Fractal Generation

The second algorithm was used to produce a Mandelbrot fractal image. This is an algorithm where more data set produced was larger than the data set used as input. The output data was used to generate the final image using a python program. Below you can see the image generated by the simulation.

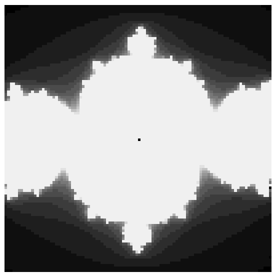

Figure 4: Fractal Image Generated by Simulation

3.2 Rasterization

Lastly, a rasterization program was attempted on the processor as well. Rasterization is the process where an image is parsed to determine what colour each pixel should have on the screen. There are many ways of carrying out the process. The algorithm used in our case was one that was better suited to stream processing architectures. The entire process is broken up into different stages, each of which is simulated on the processor. This is one of the better examples that the xStream processor is fit to use for real world applications.

4 Discussion and Conclusions

The stream processing architecture utilized in the xStream processor maximizes arithmetic intensity (defined as operations per memory access) by applying the same set of instructions to data. However, a high bandwidth requirement still exists given the parallel nature of the processor. Using a memory routing network that expands well and provides the required bandwidth is an important aspect of the xStream design. It is also important to be able to hide the latencies involved with transferring data. Hence, single cycle multithreading was implemented.

The xStream instruction set has also been designed to ignore certain issues that are usually addressed by many processors. By putting the conditional statements on the load/store instead of on the jump, stalls are no longer a concern. When a load or store doesnt execute, the next data set from memory is loaded.

The design of a fully functional and parameterizable floating point unit was also a challenge. As outlined below, there is still some work that could be done to improve some of the components of the FPU.

5 Future Work

There are a number of future improvements that can be made to the xStream processor architecture. The division and square root instructions are the two bottlenecks in terms of both area and performance. A variable length pipeline could be used to enhance the performance of the system. The square root and division operations could be redesigned to utilize more clock cycles per instruction. This enhancement would allow the performance of other common operations to be improved. Both instructions could be designed to share hardware resources and use algorithms such as high-order radix SRT or reciprocal multiplication in combination with lookup tables to improve performance at the expense of area.

Furthermore, a fully-functional external memory subsystem could be implemented to create a complete processing platform. The external memory system could support irregular memory accesses such as texture fetches in computer graphics. However, this approach would require significant changes to the existing memory controller.

References

[1] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, B. Khailany, "The Imagine Stream Processor," *Proceedings of the International Conference on Computer Design*, 2002, pp. 282-288.

[2] N. Jayasena, M. Erez, J. H. Ahn, and W. J. Dally, "Stream Register Files with Indexed Access," *Proceedings of the Tenth International Symposium on High Performance Computer Architecture (HPCA-2004)*, 2004.

[3] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany, "Efficient Conditional Operations for Data-Parallel Architectures," *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, 2000, pp. 159-170.

[4] S. Rixner, et al, "Register Organization for Media Processing," *Proceedings of the Sixth Annual International Symposium on High-Performance Computer Architecture*, 2000, pp. 375-386.

[5] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, B. Towles, "Stream Scheduling," *Proceedings of the 3rd Workshop on Media and Streaming Processors*, Dec. 2001, pp. 101-106.

[6] K. Sankaralingam, S. W. Keckler, W. R. Mark, D. Burger, "Universal Mechanisms for Data-Parallel Architectures," *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2003, p. 303.

[7] T. Aila, V. Miettinen, P. Nordlund, "Delay Streams for Graphics Hardware," *ACM Transactions on Graphics (TOG)*, July 2003, pp. 792-800.

[8] E. Lindholm, M. Kilgard, and H. Moreton, "A User-Programmable Vertex Engine," *Proceedings of SIGGRAPH*, 2001, pp. 149-158.

[9] W. R. Mark, K. Proudfoot, "The F-Buffer: A Rasterization-Order FIFO Buffer for Multi-Pass Rendering," *Proceedings of Graphics Hardware*, Aug. 2001, pp. 57-64.

[10] N. A. Carr, J. D. Hall, and J. C. Hart, "The Ray Engine," *Proceedings of Graphics Hardware*, Eurographics Association, Sep. 2002, pp. 37-46.

[11] J. Fender and J. Rose, "A High-Speed Ray Tracing Engine Built on a Field-Programmable System," *Proceedings of the IEEE International Conference on Field-Programmable Technology*, Dec. 2003, pp. 188-195.

[12] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *Proceedings of SIGGRAPH*, 2004.

[13] M. McCool, S. Du Toit, T. Popa, B. Chan, K. Moule, "Shader Algebra," *ACM Transactions on Graphics (TOG)*, Aug. 2004, pp. 787-795.

[14] M. McCool, Z. Qin, T. S. Popa, "Shader Metaprogramming," *Proceedings of Graphics Hardware*, Eurographics Association, Sep. 2002, pp. 57-68.

[15] M. McCool, S. Du Toit, *Metaprogramming GPUs with Sh*, AK Peters Ltd., 2004.

[16] Pavle Belanovic and Miriam Leeser, "A Library of Parameterized Floating Point Modules and Their Use" *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*. Sep., 2002.