Extending a Stream Programming Paradigm to Hardware Accelerator Platforms

Sek M. Chai
Motorola, Inc.
Schaumburg, IL, USA
sek.chai@motorola.com

Nikolaos Bellas
Computer Engineering and Communications Department
University of Thessaly, Volos, Greece
nbellas@uth.gr

Abelardo Lopez-Lagunas
Departamento de Electronica y Control
ITESM-Toluca, Mexico
abelardo.lopez@itesm.mx

ABSTRACT

High performance stream processing is computationally intensive and localized, performing the same operations on many data elements. These characteristics lend themselves for efficient hardware acceleration. In this paper, we describe a streaming programming model in which computation and data communication are explicitly separated and optimized. We then briefly describe hardware accelerators using the streaming programming model in two forms: an SoC coprocessor called RSVP (Reconfigurable Streaming Vector Processor), and a synthesizable hardware accelerator using our architectural synthesis tool called Proteus. We provide a retrospective of performance, cumulating more than eight years of research in streaming hardware accelerators.

1. INTRODUCTION

Streaming processors are becoming a mainstream paradigm being applied to high performance computing. In certain applications, streaming is becoming a necessary element to satisfy the computations' needs such as low latency and high throughput. The computing model may also lend itself to efficient energy consumption with more efficient handling of memory accesses.

Stream processors operate on data sequences using localized computation kernels. The memory access can be deterministic such that computation can hide memory latency with data prefetching. These characteristics, in turn, expose data parallelism for the computing platform, enabling a compiler to schedule data movement apart from the computation. By operating on large sets of data, the streaming computation reduces instruction overhead and hardware accelerator setup time.

This paper is structured as follows. In Section 2, we argue for a streaming programming model whereby the computation and communication are explicitly defined by the programmer. In Section 3, we present the stream hardware accelerators, including an SoC coprocessor and a toolset allowing automatic hardware synthesis. Section 4 summarizes our observations about these accelerators as well as the toolset development.

2. STREAMING PROGRAMMING MODEL

There are several programming languages that describe how computation should be performed in streaming architectures. Some examples are StreamIT[1], Brook[2], and Streamware[3]. We advocate a stream programming paradigm with separation of concerns between the tasks related to computation and communication. This separation of concerns decomposes the software into manageable and comprehensible parts where we can more easily identify and expose areas for performance improvement. Presently, we use a stream programming model that allows a programmer to explicitly define the data streams between computation kernels or from memory. Tasks such as data loads, stores, and alignment are assigned to dedicated mechanisms called stream units. Tasks related to the actual computation are grouped as a kernel and assigned to dedicated hardware mechanisms called datapaths. The datapath consists of functional units with a flexible interconnection network [4,5]. Figure 1 illustrates an architectural template consisting of stream units and datapath.

The stream units make use of data prefetching and alignment techniques to move data elements ahead of the computation and arrange them in the order needed by the datapath. This is crucial to keep the functional units busy; otherwise the performance gains would not be as significant.

A kernel of computation is a set of localized processor operations that are independent and self-contained. The processing in each kernel is regular or repetitive, which often comes in the form of a loop structure. These computation kernels can operate without frequent external interactions with other kernels. Global variables are usually not referenced in a kernel. Instead, the stream and other scalar values, which hold persistent state, are identified explicitly as variables in a data stream or as signals between kernels. Kernel regularity, in turn, produces uniform memory access of data elements (or data streams). Stream data appear to be sequential to the computation kernels even though data is usually scattered throughout memory.

The programmer defines a computation kernel using a streaming data flow graph (sDFG) language [6]. A sDFG consists of nodes, representing basic arithmetic and logical operations, and directed edges representing the dependency of one operation on the output of a previous operation. Each node is denoted by a descriptor, which specifies the following: input operands, the operation, the minimum precision of its output value, and the signedness of the output result.

To express memory access or communication tasks, the programmer uses stream descriptors as an application programming interface (API) to express the shape and location of data in memory [7]. Within this paradigm, we define the term *stream element* as the individual datum for processing. For image processing applications, pixel values in an image are the usual stream elements. A group of stream elements makes up a *stream record*. The grouping and the order of the stream elements in the stream record corresponds to the preferred alignment of the computation kernel. A stream descriptor is represented by a seven tuple field consisting of the following: a *start_address* of the first element in a stream record, a *stride* value indicating the spacing in stream elements between two consecutive elements in the stream record, a *span* value indicating the number of stream elements that are gathered before the skip offset is applied, a *skip* value indicating the displacement in stream elements that is applied between groups of span elements, a *type* indicating how many bytes are in each element, and a *stream_count* value for the number of elements in the stream record.

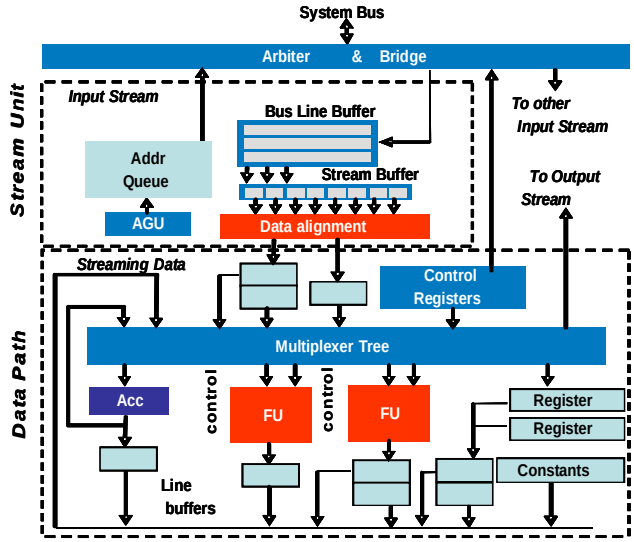

Fig 1. Streaming accelerator template with Stream Unit and Data Path

3. Stream Hardware Accelerators

Our hardware architecture is based on a streaming accelerator template consisting of stream units and a datapath. Stream units transfer data to and from the memory system, while the datapath performs stream computation on the data. The accelerator runs alongside the host processor instruction flow, without participating with its control flow. Software synchronization for data coherence is needed for the accelerator to operate asynchronously with the host processor.

Reconfigurable Stream Vector Processor

RSVP™ is an SoC implementation based on the streaming accelerator template. The first chip has been fabricated using TSMC 0.18µm CMOS technology, it has 9.5-M transistors and an area of 5.04 × 9.03 mm^2. It is integrated into an ARM946ES-based SoC with a complete set of peripherals for an embedded camera. A second generation design was completed but not fabricated. Readers are referred to [4,6] for more information.

The use of RSVP has shown performance gains that range from two to twenty times when compared to a conventional scalar core. Originally targeted for multimedia processing, the application suite consists of image processing, video codec and other computer vision algorithms.

There are several development tools that facilitate application development on the RSVP architecture. Among those tools is a cycle-accurate simulator of a system that includes RSVP, an ARM scalar core, and a memory controller which is capable of simulating the behavior of the memory hierarchy, including latency from the bus and most DRAM modules. The cycle accurate simulator can collect metrics on the data transfers of the scalar core, RSVP, and memory controller.

Proteus Toolset

The Proteus toolset [5] uses a template-based approach to produce area-efficient hardware designs based on the constraints of the application, user requirements, and system constraints. The processor template consists of a memory mapped hardware accelerator consisting of stream units and a datapath.

The following describes the Proteus design flow. First selected kernels of the sDFGs and stream descriptors are used to allocate a set of functional units (within resource constraints set by the user). Then the computation sequence, resembling a series of VLIW (very long instruction word) instructions, is defined using modulo scheduling so that the processing can operate properly. Next, an interim hardware description file is created to list the components within the accelerator. A set of state machines is also created to generate the proper control signals for each functional unit. The hardware description (Verilog) is then generated and synthesized into an FPGA. This process is repeated for each streaming kernel.

The stream unit design is also generated based on processing needs, in addition to user and system constraints. The size and number of buffer elements are chosen to meet the performance of the bus as well as the target performance of the generated datapath. For example, the size of the bus address queue is set so that bus transfers are sustained without stalling the datapath. The number of line buffers used to store data is set so that the stream unit can buffer the proper number of elements consumed by the datapath in each cycle.

We have shown performance improvements of Proteus generated hardware accelerators for several applications [8]. As an example, Table 1 shows the execution time of a lens distortion correction algorithm in a software implementation on a 2.5 GHz Core 2 Quad processor and FPGA hardware running on the Virtex-4 LX-80, 62.5 MHz FPGA [9]. For this application, Proteus generated 100K lines of synthesizable Verilog and appropriate testbench from 800 lines of sDFG code.

Table 1. Performance comparison between Core 2 Quad and Proteus generated hardware accelerator

Design	Frame rate	Speedup over SW	Speedup per Hz	Speedup per Hz per thread
SW	5.26	1	1	1
HW	22	4.18	167.2	668.8

4. Result Summary and Analysis

This section presents a brief retrospective review of the performance and design of streaming accelerators. It is presented with the hindsight of what was learned from the RSVP and Proteus projects.

Programming Model

With the separation of concerns between communication and computation, we find that software complexity and comprehensibility can improve. When the programmer explicitly expresses the data access and communication, we can generate the appropriate memory structures such as buffers and bus networks that implement the memory subsystem. In addition, the datapath is simplified without complex load/store and memory stall logic.

We also find aspects of reuse and faster algorithm development because the sDFG is free from aspects related to the memory infrastructure. That is, the computation is free from elements related to operation hoisting and other manipulations due to memory latency.

The familiar single-core programming model is well accepted and preferred by programmers. However, most programmers have an initial adverse reaction to sDFG as a non-C/C++ programming language. As such, we are continuing our efforts for compiler generated sDFG to further facilitate programming in our stream programming model.

Memory Bandwidth

In our streaming paradigm, we are effectively dedicating chip-area (or transistors) to the movement of data using memory address generators rather than to the buffering of data in large caches. Our experiments [4,6,7,8,9] show that data prefetching is

an effective mechanism to take advantage of available bandwidth in between peak access periods. The performance becomes dependent more on average bandwidth of the memory subsystem and less sensitive to peak latency of any memory access. There is an overall reduction in processor stalls due to slow memory access, which also help alleviate memory wall issues that limit today's traditional architectures.

We find that stream descriptors are reasonable means to express stream data of varying complexity. Because they are not dependent on compiler manipulation and discovery of data dependencies, stream descriptors are better able to express the actual movement of data because the programmer directly states the actual grouping of data. There is no dependence on a compiler to deconstruct complex nested loop structures to find an optimal scheduling of data transfers.

We have explored different means to describe harmonic sequences of memory accesses and have plans on extending the stream descriptors with the addition of an *Offset* parameter consisting of a user defined function that computes the dynamic change in shape and location of the next stream record. We are planning to extend our application suite to database and scientific algorithms.

Stream Unit

The stream unit design can be very complicated since it handles every possible access from the datapath and all data shapes described in the stream descriptors. An ideal design would represent a hardware implementation similar to that of a large multiport register file which is difficult to scale. We find that we can optimize the stream unit design based on the data shape described by the stream descriptors and based on the maximum set of stream elements required by the datapath.

Our experiments show a need to have a sufficiently high performance memory subsystem and bus protocols [7]. Modern bus protocols and advanced streaming memory controllers that can support multiple pending requests are needed to sustain the throughput of the hardware accelerators.

ISA extensions

The interconnection network within the datapath allows for the formation of deep pipelines. However, if the type of functional units are not selected correctly, the hardware resources quickly become scarce because the compiler is not able to schedule timely operations on them. The result would be longer execution time with many NOPs in the VLIW slots. To address this issue, profiling is done on the application suite in order to find the proper mix of functional units for the SoC design. For a reconfigurable hardware platform such as FPGA, the specific set of functional units for that algorithm can be selected instead.

For certain imaging related applications, we have found the need to increase the size of internal buffers to hold intermediate data. These buffers form internal look-up-tables allowing a more flexible data-dependent access.

We have found larger speedups for larger kernels. This is because the setup time required to configure the hardware is better amortized over longer kernel executions. As such, we have explored the use of merging multiple kernels to reduce the effect of the setup overhead.

FPGA cells

The distributed SRAMs and logic resources in an FPGA make it amenable for a system using streaming hardware accelerators. With the streaming paradigm, there is no dependence on large caches to get higher performance. Instead, stream units are used to move data efficiently through the memory subsystem.

On the other hand, the logic cells consisting of four or six input LUT (look up tables) make it less efficient to map wide functional units, especially those with irregular bit-widths (required for precision). Furthermore, multiplexer structures for the interconnection fabric become a significant consumer of LUTs with larger kernels requiring larger number of functional units.

5. CONCLUSIONS

Streaming hardware accelerators are as compelling as they were earlier this decade because there are still applications demanding high performance and efficient computing platforms that can not be delivered with today's processors. The RSVP and Proteus-generated streaming hardware accelerators were successfully designed and used with our stream programming paradigm.

For the RSVP and Proteus projects, we took a conservative approach to rely on the programmer to explicitly define the computation separately from the data movement. Our intent was to accelerate an application using a familiar single-core programming model, and at various points in time, we had considered integrating a more advanced compiler infrastructure to bridge the programming gap. Because of several reasons, such as financial support, we did not increase the scope of the project to include compiler technology. We have also found that the barriers for adopting new languages or models of computation are large. Other contributions related to our core architectural issues such the separation of concerns and mechanisms to hide memory latency with deep pipelining have allowed us to quickly develop state of the art product prototypes, ranging from smart cameras to video teleconferencing.

6. ACKNOWLEDGEMENTS

The authors acknowledge previous many contributions by RSVP™ and Proteus design team at Motorola Labs.

7. REFERENCES

[1] W. Thies, et. al., StreamIT: A Language for Streaming Applications. Intl Conf on Compiler Construction, *Lecture Notes in Computer Science*, vol.2304, Springer-Verlag, London, 179-196

[2] I. Buck, et. al., Brook for GPUs: Stream Computing on Graphics Hardware, ACM SIGGRAPH 2004 Papers, *SIGGRAPH'04*, ACM, New York, NY, 777-786

[3] S. Wei Liao, et. al., Data and Computation Transformation for Brook Streaming Applications on Multiprocessors, CGO, 2006

[4] S. Chiricescu et. al. The Reconfigurable Streaming Vector Processor, RSVP™, *MICRO-36*, Dec. 2003, 141-150

[5] N. Bellas, et. al. Template-based generation of streaming accelerators from a high level representation. *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2006, Napa Valley, CA

[6] S. M. Chai, et. al., Streaming Processors for Next Generation Mobile Imaging Applications, *IEEE Communications Magazine*, vol 43, no 12, Dec 2005, 81-89.

[7] A. López-Lagunas, S. Chai, "Streaming Data Movement for Real-Time Image Analysis", *Journal of Signal Processing Systems*, published online, January 22, 2009. To appear in a Special issue on Computer Architecture for Real Time Analysis.

[8] N. Bellas, et. al., FPGA Implementation of a License Plate Recognition SoC using Automatically Generated Streaming Accelerators. *13th Reconfigurable Architecture Workshop (RAW)*, April 2006, Rhodes, Greece

[9] N. Bellas, et. al., Real-Time Fisheye Lens Distortion Correction Using Automatically Generated Streaming Accelerators, *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2009, Napa Valley, CA