

Exploring Irregular Memory Accesses on FPGAs

Robert J. Halstead
CSE Dept.
UC Riverside
Riverside, California 92521
rhalstea@cs.ucr.edu

Jason Villarreal
Jacquard Computing
Riverside, California 92507
jason@jacquardcomputing.com

Walid Najjar
CSE Dept.
UC Riverside
Riverside, California 92521
najjar@cs.ucr.edu

ABSTRACT

Algorithms that exhibit irregular memory access patterns are known to show poor performance on multiprocessor architectures, particularly when memory access latency is variable. Many common data structures, including graphs, trees, and linked-lists, exhibit these irregular memory access patterns. While FPGA-based code accelerators have been successful on applications with regular memory access patterns, they have not been further explored for irregular memory access patterns. Multithreading has been shown to be an effective technique in masking long latencies. We describe the compiler generation of concurrent hardware threads for FPGAs with the objective of masking the memory latency caused by irregular memory access patterns. We extend the ROCCC compiler to generate customized state information for each dynamically generated thread.

Categories and Subject Descriptors

B.4.4 [Input/Output and Data Communications]: Performance Analysis and Design Aids—*Formal Models*

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

Algorithms exhibiting irregular memory accesses are notoriously difficult to parallelize. Because of poor locality they do not benefit from caching. Their performance degrades considerably when run on NUMA multiprocessors. Multithreaded architectures, where memory latency is masked by the rapid switching between independent concurrent threads, have been shown to be particularly adapted to these algorithms.

Field Programmable Gate Arrays (FPGAs) are well known for their speed and efficiency on regular algorithms that operate on massive data sets. In an FPGA-based hardware acceleration the most frequently executed computation(s)

is synthesized as a customized data path through which the data is streamed. Applications that have been demonstrated to benefit from FPGA acceleration include signal and image processing, computer vision, data mining, bioinformatics, financial analysis, etc.

This paper describes a first attempt at a compiler generated hardware multithreading execution on FPGAs. We extend the ROCCC compiler toolset to support the initiation of multiple long latency memory accesses. The compiler generates the necessary hardware structures to synchronize the results with the related threads. The paper is organized as follows: Section 2 summarizes the related work. The ROCCC toolset is described in Section 3. The implementation of the compiler generated multithreaded execution on the FPGA is described in Section 4. The results of the experimental evaluation on the Convey Computers HC-1 are reported in Section 5.

2. RELATED WORK

2.1 Multithreaded Architectures

In the late 1980's research into multi-processor systems with large shared memory was being conducted. The Horizon architecture [6, 9] was built with 256 custom processors. Research showed an average of 50-80 clock cycles per memory accesses, and most all memory request were completed within 128 cycles. The processor in the Horizon architecture was thus built to manage the state information for 128 concurrent thread. Hence support up to 128 outstanding memory requests masking the memory latency caused by having 256 processors sharing a common memory.

In the early 90's the Tera Corporation, building upon the experience acquired with the Horizon machine, built the Tera MTA [2, 1]. The MTA design consisted of 256 processors sharing 64 GB of memory organized as a distributed NUMA architecture. Its interconnection network allowed better scaling to a larger number of processors. It also forced instruction requests through a shared cache lowering the network traffic. Custom processors supported the issuing of one memory request per thread per cycle. The maximum memory latency from any processor to any memory module was 128 cycles. Each processor could support up to 128 active threads. The MTA design [8] was later evolved into the Cray XMT [4]. While the MTA still had 256 processors the XMT machine has 8192 processors. The shared memory was also increased from 1TB to 128TBs for the MTA, and the clock speed was improved from 220MHz to 500MHz.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IAAA'11, November 13, 2011, Seattle, Washington, USA.

Copyright 2011 ACM 978-1-4503-1121-2/11/11 ...\$10.00.

2.2 Heterogeneous Platforms

The 1980’s also saw reconfigurable fabrics being integrated into large supercomputers. These include a large number of cutting edge CPUs coupled with a number of FPGA devices with full or partial sharing of memory. Notable among these is the Cray XD1. The Cray XD1 evaluated by the Naval Research Laboratory [7] consisted of 432 dual-core processors with 144 Virtex-II FPGAs and six Virtex-4 FPGAs. The machine consisted of 150 nodes each with one FPGA, two processor cores and 8GBs of shared memory. Of these, 144 had one Virtex-II, and another six had one Virtex-4 FPGA.

The Convey Computers HC-1 [3] is the first heterogeneous machine to support cache coherent shared virtual memory accesses from both the software (CPU execution) and the hardware (FPGA execution). This virtual memory allows an application to switch execution between software and hardware. Without the need to offload data this switch can be made with little overhead. The HC-1 has four Virtex-5 LX330 FPGAs further allowing multiple sections of an application to be written to a FPGA without need of reconfiguration at runtime. In the HC-1ex the Virtex 6 LX760 is used instead of the Virtex 5.

3. ROCCC TOOLSET

ROCCC [10, 5] is a C to VHDL compiler toolset specifically designed for the generation of FPGA-based code accelerators. Its distinguishing features are its extensive set of compiler transformations and optimizations. ROCCC was not designed to create hardware for entire applications, but instead focuses on the critical regions of large software systems. The critical regions typically consist of a loop nest performing extensive computation on large amounts of data. Hence, the ROCCC code takes advantage of the extensive amount of parallelism available on FPGAs and the ability to implement large computational pipelines on streams of data while attempting to minimize off-chip memory fetches and control flow, which are better handled on microprocessors.

Among the ROCCC design goals: maximize throughput, minimize memory accesses, minimize the size of the generated circuit, support code reuse through the import of modules in C, VHDL or as IP cores, generate platform independent code and support fast design space exploration.

In its current design ROCCC supports codes that have memory accesses whose order is compile-time determinable. These can be in one, two or N dimensional arrays. In this paper we describe an extension to ROCCC that supports irregular memory accesses. This extension is currently designed for the Convey HC-1 but could be extended to other

Algorithm 1 Summation written for ROCCC

```

void summation(int **A, int *B, int *C,
               int m, int p) {
    int i, j;

    for(j = 0; j < m; ++j) {
        for(i = 0; i < p; ++i) {
            C[j] += A[j][B[i]];
        }
    }
}

```

platforms that can support multiple outstanding memory request and where the masking memory latency can be beneficial.

4. IMPLEMENTATION

In this section we first describe the general model of latency masking threads in hardware. Next we describe our test application that requires irregular memory accesses and then the extensions made to the ROCCC compiler to support this application.

4.1 Latency Masking Threads

Using concurrent threads to mask memory latency is not a new idea. As memory accesses became the bottleneck of many architectures they moved to models that can handle multiple outstanding memory requests for masking the latency. Traditionally, FPGAs circuits have taken a slightly different approach. Circuits are designed as pipelines handling data in a predefined order. Knowing the order allows the data to be fetched beforehand and streamed directly as needed. But this streaming approach cannot be applied to circuits requiring irregular memory accesses. Concurrent threads is an unexplored option for these designs. However, with much slower clock frequencies than CPUs and GPUs it is vital to mask long memory latencies when considering performance in spite of the FPGAs massive parallelism.

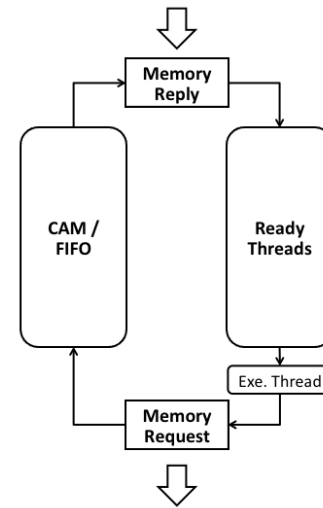


Figure 1: Multithreaded execution support framework. Threads will wait in either a CAM accessed with a unique id, or a FIFO ordered as threads make memory requests.

In the proposed multithreading framework, as shown in Figure 1, each thread is suspended after a memory read operation is performed. Outgoing requests are assigned a unique tag, and the thread waits in a CAM until data is returned. This approach requires the memory system to return data with the appropriate tag to index the CAM. State information about the thread can be stored in the CAM directly, or as a pointer to on-chip memory with the state data. While this is not the only model, as will be elaborated next, it is the most general.

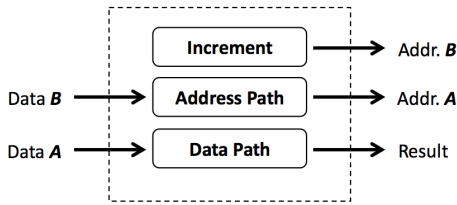


Figure 2: Summation of a single row. Values are first fetched from B . Processing these values determines a new location in A . The value stored at this location is accumulated into the final result. This is repeated for each row in A .

Optimizations can be applied to the framework if an architecture permits. A CAM is necessary when returned data could arrive out of order. If the architecture the FPGA is in supports *in order* memory access we can replace the CAM with a FIFO data structure. The FPGA can dispense with the memory tags because threads could be pushed into the FIFO based on the order of memory requests. Systems like the Convey HC-1 can be made to reorder data returned from memory, and have been done so for the experiments done in this paper.

4.2 Example Application

The example we use to describe the proposed system is expressed by Equation 1 and its code shown in Algorithm 1. It takes a two dimensional array $A[m, n]$ of values, and a list of indexes $B[p]$ as input. Where all values in B are less than n . The application runs through the rows, m , of A summing the values at the indices provided in B and storing each result into the corresponding element of C .

$$C[m] = \sum_{i=1}^p A[m, B[i]] \quad (1)$$

For this paper we implemented a summation data-path, but the it could have, just as easily, been a comparator or any other synthesizable data-path. The goal of this paper is not the data-path generation, but the generation of VHDL for irregular memory accesses. The sample application requests addresses into A based on data retrieved from B .

4.3 ROCCC Extension

The ROCCC compiler is modified to detect and support irregular memory accesses. In its traditional use, the ROCCC compiler instantiates an address generator, based on the loop and array parameters, that issues memory addresses. The returned data is treated as a stream. In our extension the address generator is similarly used however the returned data is a stream of addresses to be resubmitted to the memory system. Furthermore, the returned data, indirectly addresses, is matched to and used by its generating *thread*. In the above example, each innermost loop constitutes a thread that generates a specific $C[j]$.

The above example is synthesized into three major components as shown in Figure 2. Knowing the size of B allows the increment component to generate indexes in sequential order. Receiving the values fetched from B the address path can generate a new index into A based on the current row

being accessed. Finally the values fetched from A enter the data-path summing a result that will be returned once all values have been processed.

Figure 3 shows the implementation where the outermost loop is unrolled allowing the concurrent evaluation of multiple values in C . The only limiting factor on the degree of unrolling of the loop is the availability of memory controllers interfacing to the memory subsystem.

4.4 Implementation on FPGA

To handle irregular memory accesses our extension connects between ROCCC’s address generation, and the data-path. Instead of data being retrieved and sent directly into the data-path it is rerouted to another address generator. This address generator requests a new memory location based on this incoming data, and the current state of the application. For the example presented in this paper the address generator will know the current row being processed, and the total number of columns n in A . The base offset is computed inside our addresses generator based on these values. As data is streamed it gets added to this offset to produce the index into A . After retrieving this data from memory it is then feed into the data-path.

With addresses being generated dynamically for A it is impossible to predict subsequent memory accesses, hence successive accesses cannot be streamed from the memory to the data path. The only form of parallelism left, in addition to the operation parallelism within the thread itself, would be by unrolling the outer loop on the rows on A . Because each row is independent we can processes multiple rows at the same time without conflicting data. We could also unroll along the columns. Fetching multiple memory locations from B and process them in parallel.

5. EXPERIMENTAL EVALUATION

Testing of our application was done on the Convey HC-1. The FPGA memory interface for this machine is limited to 150 MHz. While a standalone version of our circuit synthesized to over 200 MHz we had to adopt the HC-1’s speed limitation. The Convey’s memory interface consists of 16 memory channels per FPGA for input or output data streams. Each memory channel supports a maximum of 256 outstanding memory requests. Our experiments use only one of the four available FPGAs.

As shown in Algorithm 1 the number of columns in A is set to the length of B : it maximizes the number of memory requests based on the memory allocated, and makes the code easier to read. We could allow A to have an independent number of columns from B by creating another parameter. The index into A would then compute its base location with the new parameter and current row. The offset would still rely on data coming from B .

We have explored the effects of unrolling our design. At

Table 1: FPGA and Software Execution Time

Data Size (10^6)	SW	FPGA		
		no unroll	unroll 2	unroll 7
1	0.08	0.08	0.02	0.02
10	0.72	0.65	0.09	0.04
100	7.23	6.24	0.78	0.25
200	13.71	12.45	1.53	0.47

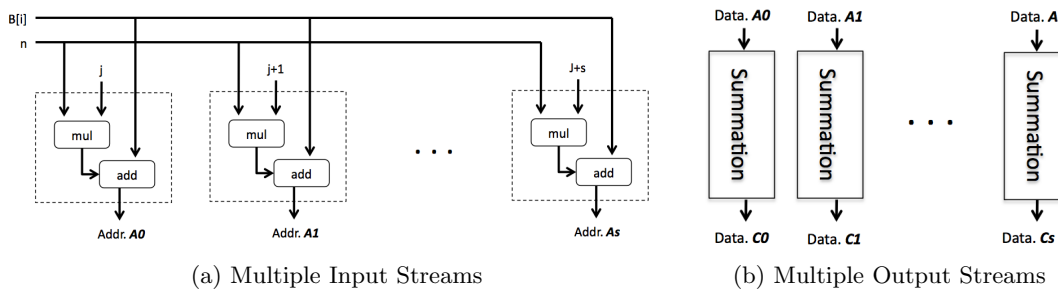


Figure 3: Unrolled implementation. When unrolling the design along j creates multiple address generators for A and multiple summation datapaths. B runs along i so its values are not unrolled, but input data is duplicated and processed by the address generators for A .

this stage of our design we do not yet support the multiplexing of multiple streams, of data or addresses, on a physical memory channel, the limiting factor is the number of memory channels. B , A and C each requires one memory channel. When the outer loop is unrolled the data from B can be shared by each instance of the loop, so the maximum unrolling is seven concurrent loop iterations (15 channels being used). We report the data for no unrolling, two and seven concurrent loop bodies.

Because the Convey machine shares the same memory structures for both hardware and software we opted to do our software implementation on the Xeon cores of the HC-1. This gives us a more accurate measure on how memory requests affect overall execution time. The HC-1 uses an Intel Xeon CPU, and Xilinx Virtex5 LX330 FPGAs.

In Table 1 we compare the execution time of software to our FPGA implementations. As expected the FPGA’s performance improves as the data set’s size increases. With 1 million elements the startup time has more effect on the overall execution time. For this reason the implementation unrolled by a factor of two was fastest. When data sets grow to 100 million elements software begins taking significant time. In this data range the FPGA is still executing under 1 sec, and the speedup for the FPGA is over 29X.

6. CONCLUSION

We have described the extensions to the ROCCC compiler toolset that can generate a multithreaded accelerator on an FPGA. We evaluate this implementation on the Convey Computer HC-1 considering the state of the cache on the overall execution time. Using only one Virtex 5 LX330 we achieve a speedup over 29X. Noticing that overall speedup on our application improves as the data size increases. The kernel code used to demonstrate the implementation and evaluate the speed-up is a very simple code. We expect that more complex applications, relying on complex memory accesses and/or having more elaborate inner loop computations, would achieve a much larger speed-up over traditional software.

Acknowledgments

This work has been supported in part by NSF Awards 0905509 and 0811416, by Jacquard Computing Inc. and by the Air Force Research Lab.

7. REFERENCES

- [1] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proc. of the 6th Int. Conf. on Supercomputing, ICS '92*, pages 188–197, New York, NY, USA, 1992. ACM.
- [2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. of the 4th Int. Conf. on Supercomputing, ICS '90*, pages 1–6, New York, NY, USA, 1990. ACM.
- [3] T. M. Brewer. Instruction set innovations for the Convey HC-1 computer. *IEEE Micro*, 30:70–79, March 2010.
- [4] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In *Proceedings of the 2nd Conference on Computing Frontiers, CF '05*, pages 28–34, New York, NY, USA, 2005. ACM.
- [5] <http://roccc.cs.ucr.edu/>.
- [6] J. Kuehnand and B. Smith. The Horizon supercomputing system: architecture and software. In *Proc. of the 1988 ACM/IEEE Conf. on Supercomputing, Supercomputing '88*, pages 28–34, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
- [7] J. Osburn, W. Anderson, R. Rosenberg, and M. Lanzagorta. Early experiences on the NRL Cray XD1. In *Proc. of the HPCMP Users Group Conference*, pages 347–353, Washington, DC, USA, 2006. IEEE Computer Society.
- [8] A. Snavelly, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz. Multiprocessor performance on the Tera MTA. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, Supercomputing '98*, pages 1–8, Washington, DC, USA, 1998. IEEE Computer Society.
- [9] M. R. Thistle and B. J. Smith. A processor architecture for Horizon. In *Proc. of the 1988 ACM/IEEE Conf. on Supercomputing, Supercomputing '88*, pages 35–41, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
- [10] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *Field-Prog. Custom Comp. Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134, may 2010.