Extending the Scalability of Single Chip Stream Processors with On-chip Caches

Ali Bakhoda and Tor M. Aamodt
University of British Columbia,
Vancouver, BC, Canada
{bakhoda,aamodt}@ece.ubc.ca

Abstract

As semiconductor scaling continues, more transistors can be put onto the same chip despite growing challenges in clock frequency scaling. Stream processor architectures can make effective use of these additional resources for appropriate applications. However, it is important that programmer effort be amortized across future generations of stream processor architectures. Current industry projections suggest a single chip may be able to integrate several thousand 64-bit floating-point ALUs within the next decade. Future designs will require significantly larger, scalable on-chip interconnection networks, which will likely increase memory access latency. While the capacity of the explicitly managed local store of current stream processor architectures could be enlarged to tolerate the added latency, existing stream processing software may require significant programmer effort to leverage such modifications. In this paper we propose a scalable stream processing architecture that addresses this issue. In our design, each stream processor has an explicitly managed local store model backed by an on-chip cache hierarchy. We evaluate our design using several parallel benchmarks to show the trade-offs of various cache and DRAM configurations. We show that addition of a 256KB L2 cache per memory controller increases the performance of our 16, 64 and 121 node stream processors designs (containing 128, 896, and 1760 ALUs, respectively) by 14.5%, 54.9% and 82.3% on average respectively. We find that even those applications that utilize the local-store in our study benefit significantly from the addition of L2 caches.

1 Introduction

Despite challenges to clock frequency scaling, continued reductions in process technology feature sizes are enabling manufacturers to put ever more transistors on a single chip. This combination has led the semiconductor industry towards architectures that expose greater amounts of parallelism to software. Stream processor architectures [18, 4, 1, 12] dedicate more chip area to ALUs than superscalar processors and are very effective for workloads that contain large amounts of data-level parallelism. Assuming that current scaling trends continue, in less than ten years, it will be possible to put thousands of 64-bit floating point ALUs on a single chip. Even for applications with abundant parallelism, keeping thousands of ALUs busy will be a challenging task as we show in this paper. Figure 1 shows that, as process technology scales and therefore number of ALUs per chip increases, the gap between *potential IPC* and the IPC achieved via "*naïve scaling*" widens. The potential IPC presented in the figure was measured assuming a perfect memory system incurring zero penalty cycles to access memory (along with the hardware configuration shown in Table 1). The "naïve scaling" numbers were measured assuming scaling is achieved by simply replicating stream processor cores similar to those employed in contemporary hardware [12][1] and connecting them using a mesh interconnection network without any L2 caches for non local-store memory accesses.

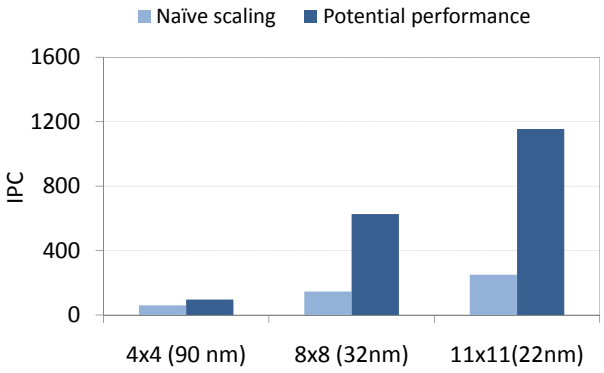


Figure 1. Potential performance assuming zero latency memory system compared to naïve scaling of cores

Although stream processors utilize silicon more effectively, they will only outperform traditional superscalar pro-

[1] We consider stream processor cores of the granularity of a Streaming Multiprocessor (SM) in the GeForce 8 Series [12].

cessors if the application contains plentiful data-level parallelism and is rewritten in a streaming model language. The significant effort this entails implies that maintaining backward compatibility among future generations of stream processors is essential to amortize programmer effort.

Recent *graphics processing units* (GPUs) are among the most successful and widely available stream processors. NVIDIA's *Compute Unified Device Architecture* (CUDA) and ATI's *Close To the Metal* (CTM) are two programming models that enable users to run data parallel kernels on recent generations of GPUs without the need to employ graphics programming interfaces as was typical for early approaches to *General-Purpose computation on GPUs* (GPGPU). Although GPUs are primarily designed for graphic applications, their shader cores resemble streaming processors. Both NVIDIA and ATI allow users to run compute kernels on the GPU's shader cores. Writing a functionally correct application is relatively easy in these environments. The challenging part is optimizing the application to take advantage of all the potential computing power the GPU has to offer. Even beyond the well known challenge of finding parallelism, there are others. For example, on the NVIDIA GeForce 8 Series, the programmer must manually change the application to avoid bank conflicts when accessing the local store since conflicting accesses can degrade performance. Achieving this can require significant programmer effort (the CUDA development environment helps by providing tools for identifying such conflicting accesses [15]).

In this paper we will study the scalability issues of a SIMD stream architecture. We use a mesh network as the interconnection fabric and then use an on chip cache hierarchy to improve the performance. The contributions of this paper are:

- We propose a multi-level cache design specially suited for stream processors.
- We quantify the effects of cache size, DRAM bandwidth and other parameters on stream processor performance.

The rest of this paper is organized as follows. In Section 2 we discuss background information about interconnection network design and memory controller limitations in stream processors. In Section 3 we discuss our proposed memory hierarchy and scalable stream processor architecture. Our experimental methodology is described in Section 4 and Section 5 presents and analyzes results. Section 6 reviews related work and Section 7 concludes the paper.

2 Background

Single chip stream processors typically contain several *stream processing cores* as well as several memory controllers integrated on a single chip. In this paper we call the stream processing cores *shader cores*—named after GPU shader cores (e.g., an SM on GeForce 8 Series hardware [12]) which are SIMD stream processors. Shader cores and memory controllers are connected using an interconnection network that is a critical component since all memory accesses go through it.

In this section, we first discuss several trade-offs for designing on-chip interconnection networks. Then we discuss the limitations of on-chip memory controllers and naïve SIMD pipeline width scaling.

2.1 On-chip interconnection networks

There are various options to design the interconnection network of a stream processor. Here we briefly discuss full crossbar, ring and mesh networks. A full crossbar provides high bandwidth, low latency and minimum latency variation. Crossbar's cost and area increase quadratically as a function of the number of nodes connected by the crossbar. Although a crossbar's cost and area are reasonably economical in small configurations, they quickly become prohibitive as the number of nodes increases [6].

A ring interconnect is used in the Cell processor [17] and ATI R600 [13]. Although a ring interconnect is not as area demanding as a crossbar, its throughput and latency become degraded as the number of nodes increases. The simulations in [2] also confirm that a 2D mesh provides higher throughput and lower latency than a ring when a network has multiple hot spot nodes (nodes with higher traffic demands). Memory controllers are the hot spots [24] in a stream processor chip with integrated memory controllers. Mesh networks are inherently scalable, but they result in variable and long latencies compared to crossbars. Since our primary goal in this paper is scalability, we opt for a mesh network. To cope with the side effects of the mesh network such as increased latency, we consider applying microarchitecture techniques to keep all the processing units busy by trying to ensure they are not starved for data. As we will show, addition of caches helps by reducing the load on the interconnect. We leave a more detailed comparison with other network topologies for future work.

2.2 Available memory controllers

Another restriction for all future processors is the number of available memory controllers per chip. According to ITRS projections [9] the number of pins per chip will not increase at the same rate as the number of transistors per chip (pin count is projected to increase at a rate of roughly 10% per year). Additionally, the number of memory controllers (total off-chip memory bandwidth) has a direct and non-negligible effect on total system cost. Inevitably, the ratio of

processing units to memory controllers is going to decrease in future stream processors. Consequently, pressure on the memory system will increase, necessitating techniques to both increase off-chip bandwidth per pin and to reduce the average number of off-chip accesses per ALU operation.

2.3 SIMD Width

One way to scale the total number of ALUs in future process technologies is to increase the number of processing elements in each shader core's SIMD pipeline (in other words designing "fatter" shader cores). This approach has drawbacks. SIMD Stream processors, such as NVIDIA's GeForce 8 series, group threads into warps for scheduling purposes [15]. A *warp* is a collection of threads that execute together in SIMD fashion on the hardware. The number of threads in a warp is equal to a multiple of the number of processing elements in the SIMD pipeline in the shader core. A warp is formed when a compute kernel is dispatched to the stream processor and afterwards all the threads in a warp execute together. The advantage of grouping scalar threads into warps is the reduction in area associated with SIMD hardware.

Increasing the number of threads grouped together, or "warp size", exposes a major performance limitations of current graphics hardware, which is branch divergence [7]. As shown by Fung et al., increasing warp size from 8 to 16 using contemporary approaches decreases throughput by roughly 30% [7] on set of non-graphics applications (similar to those we study in this paper). Furthermore, although it is possible to write applications that are aware of the warp size of the hardware they are executing upon (which is necessary on current hardware when synchronization operations are employed [15]), backward compatibility could be maintained trivially in future designs by increasing the number of shader cores instead of increasing the warp size. However, this option increases the number of nodes in the interconnection network, thus requiring a scalable interconnect such as a mesh.

3 Design and Implementation

In this section we first describe our baseline architecture, which scales to future process technologies by simply increasing the number of shader cores and using a scalable mesh interconnection network. Then we describe our modified architecture which incorporates second level caches at the memory controllers to enhance scalability. Figure 2 depicts a high level view of the stream processor architecture we consider. The portions labeled "L2 $" correspond to the proposed hardware changes.

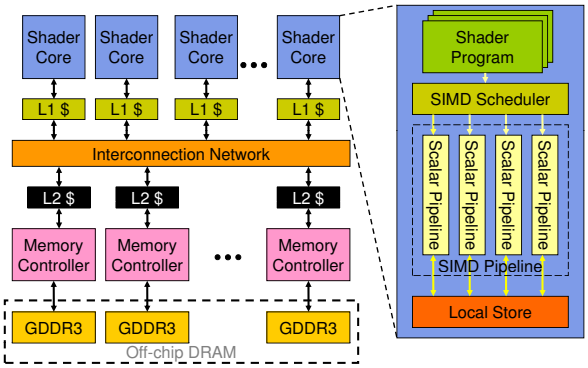


Figure 2. Streaming processor architecture overview

3.1 Baseline Architecture

The stream processor is treated as a co-processor that a CPU can offload highly data-parallel compute kernels on to. The stream processor consists of several compute nodes (labeled "Shader Core" in Figure 2) and memory nodes (labeled "Memory Controller"). Each shader core has a warp size and SIMD width of 16, and uses a seven-stage, in-order pipeline. The processing elements in a shader core share a low latency 16KB local store that is explicitly managed by the programmer. Each shader core also has a private L1 cache to back up the local store. This cache is implicitly managed by hardware similar to a traditional L1 cache.

Each on-chip memory controller interfaces a single GDDR3 DRAM chip[2] module with 4 banks.

The on-chip interconnection network can be designed in various ways. As discussed in Section 2 we use a mesh network to achieve scalability. We show that it performs reasonably well for the massively parallel benchmarks that we study.

Thread scheduling is performed with zero overhead on a fine grained per cycle basis. Each cycle a warp that is ready for execution is selected by warp scheduler and issued to the SIMD pipelines. We use a scheduling policy called DFIFO [7] which is basically a round robin technique, but if there is a cache miss in a particular warp, then that warp is not considered for the scheduling and next ready warp is selected (i.e., the round robin order can change). All the threads in each given warp execute the same instruction with different data values simultaneously in all pipelines. Whenever any thread inside a warp faces a long latency operation such as a cache miss, all the threads in the warp are taken out of the scheduling pool until the long latency operation is over. Meanwhile other threads that are not waiting are sent to the pipeline for execution. Since there are many threads running in the same shader core long latency operations can be

[2]GDDR3 stands for Graphics Double Data Rate 3 [21]. Graphics DRAM is typically optimized to provide higher peak data bandwidth.

tolerated to some extent.

We use the *immediate post-dominator* (PDOM) [7] mechanism to allow for diverging control flow among threads within a given warp. This mechanism employs a stack to allow separate traversal of different control flow paths when the threads in a single warp wish to take different paths following a conditional branch. Fung et al. [7] describe the PDOM mechanism in more detail.

It must be noted that using the local store is crucial to achieve high performance on GPUs. Using the local store alleviates the DRAM bandwidth bottleneck that many applications generally face [20]. In our designs the local store is backed up by an on-chip cache hierarchy to mitigate the effects of increased latencies incurred by the interconnection network and the extra pressure on the memory system. Each shader core also includes a 32 KB data cache for memory accesses to the non local-store "global" address space[3].

We find this baseline architecture scales well for applications with very high arithmetic intensity (ratio of arithmetic operations to memory accesses), but less well for others. Next we consider how to add additional cache capacity to reduce interconnection network bandwidth requirements.

3.2 Extending Stream Processor Scalability

Increasing cache capacity can reduce memory bandwidth requirements for applications that contain sufficient locality. We propose to achieve this by adding a second level, shared cache to the design described in Section 3.1. A shared cache can be advantageous in that some threads may not require as much capacity at any given time as others, and furthermore there may be inter-thread temporal locality among threads since they are from the same application.

For our study, none of the benchmarks require communication between threads operating on different shader cores, and all inter-thread communication for threads scheduled on a particular shader core occurs via the local-store. Hence we do not model any cache coherence protocol for our L1 or L2 caches (we leave this to future work).

We locate each bank of the L2 cache with a memory controller. The L2 caches are simple single-port set associative caches. Each L2 cache, only caches the data that maps to its corresponding memory controller and DRAM. The trade-off of L2 cache addition will be discussed in Section 5. We will show that relatively small L2 caches can substantially improve streaming application performance. Note that since L2 caches are located on the memory controller side of the network, accesses to L2 caches must always traverse the network.

[3]These memory accesses correspond to to *global memory* access in CUDA–e.g., `ld.global` and `st.global` in the PTX instruction set [16].

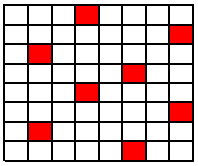


Figure 3. Layout of 8x8 configuration (shaded areas are memory controllers)

4 Methodology

To model the architecture described in this paper we extended the simulator used by Fung et al. [7] that models various aspects of a massively parallel architecture with highly programmable pipelines similar to contemporary GPU architectures. The version of this simulator used in this study employs the SimpleScalar PISA [3] instruction set to simulate scalar threads. The timing simulator accounts for how these would be grouped into warps by the hardware we model.

Table 1 shows the simulator's configuration. Rows that have multiple entries show different configurations that we have simulated. We used a detailed interconnection network model to simulate the mesh network. The interconnection network of the simulator is an extension of the simulator introduced in [6] and is highly configurable. Table 2 shows the interconnection configuration used in our simulations.

The benchmark applications used for this study were selected from SPLASH-2[23], and several CUDA programs published by NVIDIA [14]. Six benchmarks (BIN, CON, IMD, MAT, SCN and SOB) use the local store and three benchmarks (BLK,BIT and LU) do not use the local store. The simulator's programming model is similar to that of CUDA[4]. A computing kernel is invoked by a spawn instruction, which signals the SimpleScalar out-of-order core to launch a predetermined number of threads for parallel execution on the GPU simulator. Note that all the spawned threads for a specific kernel are running the same group of instructions on different data elements although they might take different control paths along the way. Our simulator supports thread blocks and synchronization instructions inside the blocks. Every time a compute kernel is spawned all the threads inside that kernel are divided into groups of threads called blocks[5]. All the threads in a block are assigned to a single shader core for running. Communication and synchronization among different blocks are not supported but threads inside each block can communicate

[4]Currently we must undergo several manual steps to prepare the benchmarks for running in our simulator which limit the number of benchmarks used in this study. In this study, we convert CUDA applications to C with meta information placed in a configuration file used to simulate the effect of launching a CUDA "grid" onto a GPU (this approach was used in [7] as well).

[5]The user specifies the number of threads in each block and hardware sets a maximum number of threads per block

Table 1. Hardware Configuration

Number of Shader Cores	8 / 56 / 110
SIMD Warp Size	16
Number of Threads per Shader Core	256
Local Storage per Shader Core	16KB
Number of Memory Channels	8 / 8 / 11
GDDR3 Memory Timing	t_{CL}=9, t_{RP}=13, t_{RC}=34 t_{RAS}=21, t_{RCD}=12, t_{RRD}=8
Bandwidth per Memory Module	8 / 16 / 32 (Bytes/Cycle)
DRAM queue capacity	32 requests
Memory Controller	out of order (FR-FCFS) [19]
L1 Data Cache Size (per core)	32KB 2-way set assoc. LRU
L1 Data Cache Hit Latency	3 cycle latency (pipelined 1 access/cycle/SIMD pipeline)
Branch Handling Method	Post Dominator [7]
Warp Issue Heuristic	DFIFO among ready warps [7]
L2 Cache Size (per Mem Controller)	256KB / 512KB / 1MB
L2 Cache Parameters	8-way set assoc. 64B lines LRU

Table 2. Interconnect Configuration

Virtual channels	4
Virtual channel buffers	16
Virtual channel allocator	islip
Alloc iters	1
Credit delay	1
Routing delay	1
VC alloc delay	1
Input speedup	2
Flit size	32

Table 3. Benchmark abbreviations

Benchmark	**Abr.**
Black Scholes option pricing	BLK
Binomial Options	BIN
Bitonic Sort	BIT
Convolution Separable	CON
Image Denoising	IMD
LU	LU
Matrix Multiply	MAT
Scan Large Array	SCN
Sobel Filter	SOB

either through the fast local store or main memory with the aid of barrier instructions[6]. All the threads that reach a barrier instruction wait for the rest of the threads in their block to catch up and, when all of threads reach the barrier, they resume execution again.

5 Performance Evaluation

To evaluate how well the designs introduced in Section 3 scale we simulated 3 different configurations. The first configuration consists of 8 shader cores and 8 memory controllers: a configuration which is intended to represent contemporary architectures. The second configuration has 56 shader cores and 8 memory controllers. This configuration has 896 ALUs (or "streaming processors" [12]) which can be integrated on a single chip by 2013 according to ITRS [9] projections. Our third configuration has 110 shader cores and 11 memory controllers. This would incorporate 1760 ALUs (streaming processors) and could be built by 2016. According to the ITRS semiconductor roadmap [9] process technology should reach 32nm and 22nm by 2013 and 2016, respectively. The number of ALUs for future generation GPUs is based on a linear extrapolation of the number of ALUs on an NVIDIA 8800 series GPU assuming constant area and overheads per shader core.

In our design the memory controllers are physically distributed over the chip. Figure 3 shows the physical layout of the memory controllers in our 8x8 configuration as shaded areas. A similar layout is applied for 4x4 and 11x11 configurations. These layouts are based on the assumption that it is possible to place pads all over the chip area so that there is no need for extra wires to connect the memory controller to chip's circumference.

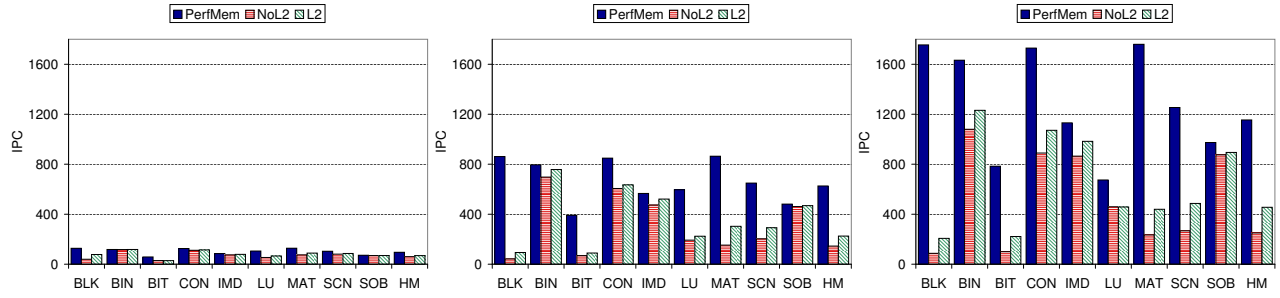
Table 3 shows the benchmarks we used for simulations along with the abbreviations that are used for each benchmark in the figures. We ran BLK and BIT for 100 million and the rest of the benchmarks for 1 billion instructions. Figure 4 shows the effect of increasing the number of cores on IPC. The first bar shows the maximum IPC for each benchmark given perfect memory (all memory accesses hit in L1 cache). The "NoL2" bar shows performance of the device without any L2 cache and "L2" shows the IPC results when a 256KB L2 cache is added per memory controller. Addition of L2 cache increases the performance by 14.5%, 54.9% and 82.3% on average for 4x4, 8x8, and 11x11 designs respectively. For the applications that use the local store (BIN, CON, IMD, MAT, SCN and SOB), the harmonic mean speedup is 5.7%, 53%, and 50% for 4x4, 8x8, and 11x11 designs respectively; for applications that do not use the local store (BLK, BIT, and LU) the harmonic mean speedup is 23.6%, 64%, and 105% for 4x4, 8x8, and 11x11 designs respectively.

Figure 5 shows L1 miss rates for all the "NoL2" and "L2" configurations in Figure 4. For some benchmarks L1 miss rates with an L2 cache are higher compared to when there is no L2. Our detailed investigation revealed that this behavior is related to a complex relationship of cache replacement (LRU) and the scheduling policy that we use to issue warps to the SIMD pipeline. When the L2 cache is added the order that memory requests return to the shader core changes dramatically; some of the requests hit in the L2 and return much faster than the ones that miss in L2 and are serviced by DRAM. This reordering effect combined with the warp scheduling policy creates this counter-intuitive behavior.

Figure 6 shows the L2 miss rates. As the number of shader cores increases from 8 (4x4) to 56 (8x8) the L2 miss rate increases for all the benchmarks. In this configuration L2 cache size remains the same as the configuration with 8 shader cores while the number of shader cores increases[7].

[6] Barrier instructions are specified by user in the body of the kernel and operate similar to CUDA's `__syncthreads()` operation [15]

[7] We hold the size of the L2 per memory controller constant consistent with the assumption the area of a memory controller block is kept a fixed ratio with respect to a shader core to minimize the impact on layout as we



(a) 4x4 (90nm) (b) 8x8 (32nm) (c) 11x11 (22nm)

Figure 4. IPC for 4x4, 8x8 and 11x11 configurations with 256KB L2 cache per shader core

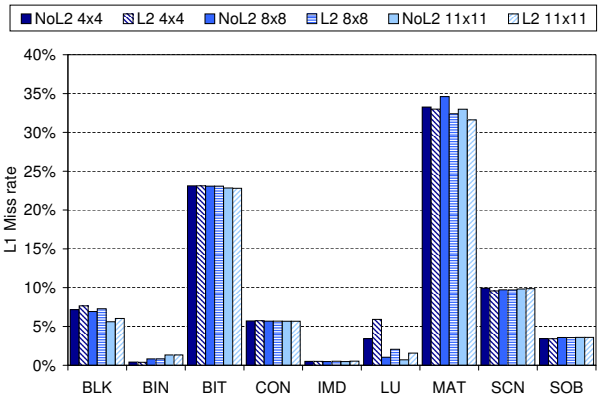


Figure 5. L1 miss rates for 4x4, 8x8 and 11x11 configuration without L2 and with 256K L2 per memory controller

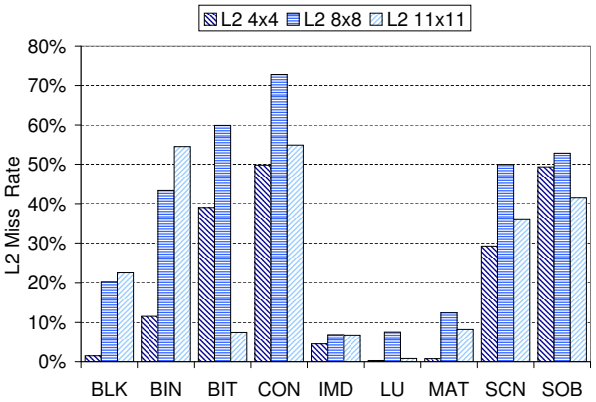


Figure 6. L2 miss rates for 4x4, 8x8 and 11x11 configurations with 256K L2 per memory controller

As the number of shader cores increases to 110 (11x11) some benchmarks behave differently and their cache miss rate does not increase despite the higher ratio of shader cores to cache capacity. This configuration has 11 memory controllers and since we add an L2 cache to each memory controller this configuration also has a higher aggregate L2 cache capacity. Our version of the LU benchmark does not have enough threads to use all 110 shader cores and therefore its L2 miss rate drops. For the SOB and BIT benchmarks the working set fits into L2 cache in this configuration. These three benchmarks are also not sensitive to L2 cache size increase as shown in Figure 8.

We measured the DRAM utilization of the various hardware configurations we model, and this data is shown in Figure 7. The utilization numbers are calculated by counting the number of DRAM responses each cycle even when there is no outstanding request to DRAM. Therefore, either few DRAM requests or poor DRAM scheduling might result in low utilization. These cases can be separated by considering the number of requests coming from the memory controllers. The first three bars for each benchmark in Figure 7 show the data for the NoL2 configurations and the next three bars show the data when an L2 cache is included. As shown in the figure, utilization for the second three bars is lower than the first three, highlighting the effectiveness of L2 cache in reducing the number of requests sent to DRAM (recall that performance increases as we add the L2 cache). Figure 7 is also consistent with Figure 6 as the DRAM utilization decreases for the benchmarks that have an decrease in L2 cache miss rate (e.g., for BLK, the miss rate increases going from 4x4 to 8x8 to 11x11, as does the DRAM utilization).

5.1 Sensitivity to Cache Size

Figure 8 shows the effects of cache size increase for 11x11 core design. Most benchmarks experience substantial performance boosts when the amount of cache per memory controller is increased from 256KB to 512KB. The improvement for increasing memory capacity from 512KB to 1MB is not as dramatic. One of the interesting observations in this graph is that benchmarks that benefit from increasing the L2 cache capacity includes those benchmarks that use the local store. BIN, CON, IMD, MAT and SCN all uti-

scale the architecture.

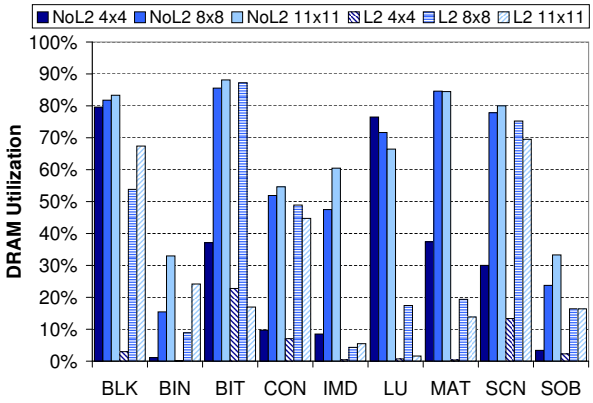


Figure 7. DRAM Utilization for 4x4, 8x8 and 11x11 configurations without L2 and with 256K L2 per memory controller

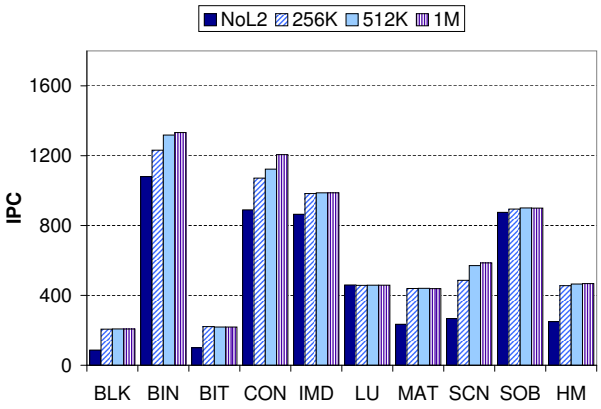


Figure 8. Sensitivity to L2 size for 11x11 configuration

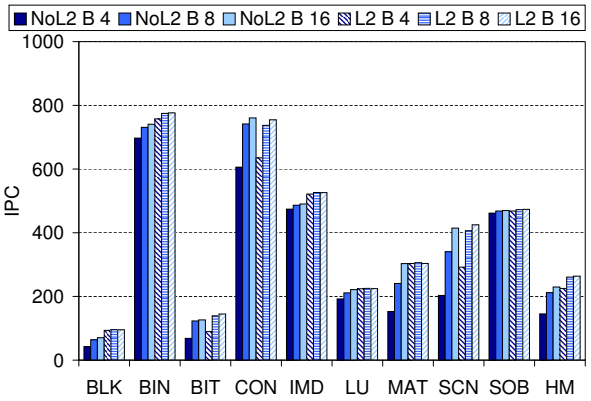


Figure 9. Sensitivity to DRAM Bandwidth for 8x8 configuration

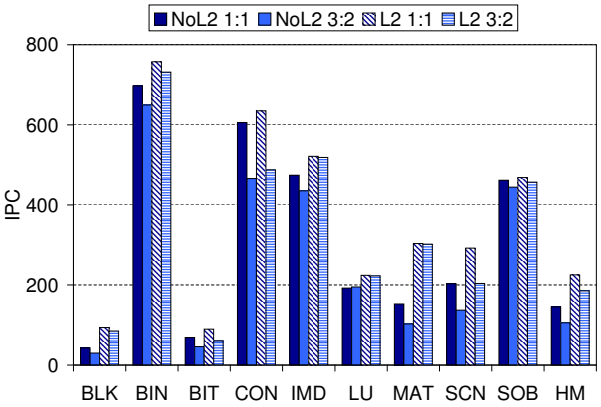


Figure 10. GPU to DRAM clock ratio effects for 8x8

lize local store and all experience a substantial performance boost as L2 cache size increases from NoL2 to 1MB. We believe the reason for this is that benchmarks that are able to easily utilize the local store have a lot of data sharing and locality—that is why we could write them in such a way that they can use local store in the first place.

5.2 Sensitivity to DRAM bandwidth

The effects of increasing DRAM bandwidth on performance are presented in this section. In order to simulate a higher bandwidth we changed the DRAM burst length while keeping the duration of an entire burst transfer (from start to finish) constant with respect to the shader core clock. For the 8x8 design with 256KB L2 caches increasing the burst length from 4 (i.e. 8Bytes/cycle) to 8 and 16 increases the performance less than 15.8% and 17.1% respectively. The same burst length increases result in 45.6% and 58% performance increases when there is no L2 cache. Figure 9 shows that the IPC of configuration with no L2 cache and burst length 16 (third bar) is only 1% higher than the design with L2 cache and burst-length 4 (fourth bar). We suspect the reason increasing burst length is not as helpful as increasing cache capacity is that not all data brought in by larger burst lengths is used before being evicted from the on-chip caches.

5.3 GPU to DRAM Clock Ratio Effects

Figure 10 shows the effects of changing the GPU to DRAM clock ratio form 1:1 to 3:2 (every 3 GPU cycles there is 2 DRAM cycles) for the 8x8 configuration. When there is no L2 cache on chip IPC drops more than 37% when clock ratio is increased to 3:2 but when a 256KB L2 cache is added to each memory controller, IPC only decreases 21%. The only counter-intuitive phenomena in Figure 10 is LU's IPC increase for about 1% when the DRAM is slowed down. This is again due to our L1 cache replacement policy: Close inspection reveals that victim L1 cache lines are selected when the memory requests comes back to L1, during the time new cache line is on its way, new requests can

still hit in the cache changing the LRU line.

6 Related Work

The Cell processor [17] is a hardware architecture that can function like a stream processor with appropriate software support. It consists of a controlling processor and a set of SIMD co-processors each with independent program counters and instruction memory. Merrimac [5] and Imagine [1] are both streaming processor architectures developed at Stanford. Merrimac is designed in a scalable way so that many Merrimac chips can be put together to form a cluster of stream processors. All these chips have on-chips stream register files (SRF). Merrimac [5] also has a single layer of cache on the memory controller side which is very expensive to access. We quantified the effects of varying cache size, DRAM bandwidth and other parameters which to our knowledge has not been published previously. All these designs have different programming models from the CUDA like programming model that we employ.

Khailany et al. [10] explore VLSI costs and performance of a stream processor as the number of streaming clusters and ALUs per cluster scales. They use an analytical cost model. The benchmarks they use also have a high ratio of ALU operations per memory reference which is a property that eases memory requirements of streaming applications. Govindaraju et al. [8] present a memory model to analyze the performance of GPU-based scientific algorithms and use it to improve cache efficiency. Their model is based on texturing hardware that uses 2D block-array representation to transfer the data between texture caches and video memory. Their design applies to previous generations of GPUs that were not directly programmable for general purpose computing.

UltraSPARC T2 [22] microprocessor is a multithreading, multi-core CPU which is a member of the SPARC family, and the successor to the UltraSPARC T1 [11]. UltraSparc come in 4, 6 and 8 core variations and each core is capable of running 8 threads concurrently. They have a crossbar between L2 and the processor cores (similar to our placement of the L2 in Figure 2). Although the T1 and T2 support many concurrent threads (32 and 64, respectively) compared to other contemporary CPUs, the number of threads is very small compared to the number on a high end contemporary GPU (e.g., the Geforce 8800 GTX supports 12,288 threads per chip). Our smallest design for example supports 8x16 threads executing in a single cycle, while it can have many more threads in the scheduler pool. As a result, the ratio of threads to cache size is much smaller in our architecture.

7 Conclusions and Future Work

In this paper, we argued that it is important that programmer effort be amortized across future generations of stream processor architectures. We focused on stream processors similar to GPUs and used a programming model similar to CUDA. We explored the benefits applications written using CUDA like models will obtain as chip resources scale. We proposed to use a mesh topology as a scalable on-chip interconnection network and to use on-chip cache hierarchy to decrease the pressure on memory system and back up the explicitly managed local store. We evaluated our design using several parallel benchmarks and showed that by adding a banked L2 cache on the memory controller side of the chip a mesh based stream processor's performance can improve substantially. Our future work includes exploring further tradeoffs in cache organization and interconnect topology of single chip stream processors. We are also planning to investigate the performance versus area tradeoff of our proposed design.

Acknowledgments

We would like to thank George Yuan, Xi Chen and the anonymous reviewers for their helpful comments. We also thank Wilson W. L. Fung for helpful discussions and the development of a significant portion of the simulation infrastructure used in this study.

References

[1] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das. Evaluating the imagine stream architecture. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 14, 2004.

[2] L. Bononi and N. Concer. Simulation and analysis of network on chip architectures: ring, spidergon and 2d mesh. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 154–159, 2006.

[3] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. http://www.simplescalar.com, 1997.

[4] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, 2003.

[5] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, 2003.

[6] W. J. Dally and B. Towles. *Interconnection Networks*. Morgan Kaufmann, 2004.

[7] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for effiecient gpu control flow. In *MICRO 40: Proceedings of the 40th annual IEEE/ACM Int'l Symp. on Microarchitecture*, 2007.

[8] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, 2006.

[9] Int'l Technology Roadmap for Semiconductors. 2006 Update. http://www.itrs.net/Links/2006Update/2006UpdateFinal.htm.

[10] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, J. D. Owens, and B. Towles. Exploring the vlsi scalability of stream processors. In *HPCA '03: Proceedings of the 9th Int'l Symp. on High-Performance Computer Architecture*, page 153, 2003.

[11] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[12] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[13] Marco Chiappetta. ATI Radeon HD 2900 XT - R600 Has Arrived. http://www.hothardware.com/printarticle.aspx?articleid=966.

[14] NVIDIA Corporation. NVIDIA CUDA SDK code samples. http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html.

[15] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 1.1 edition, 2007.

[16] NVIDIA Corporation. *PTX: Parallel Thread Execution ISA*, 1.1 edition, 2007.

[17] D. Pham, S. Asano, M. Bolliger, M. D. , H. Hofstee, C. Johns, J. Kahle, A.Kameyama, J. Keaty, Y. Masubuchi, D. S. M. Riley, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. W. D. Wendel, T.Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE Int'l*, pages 184–592 Vol. 1, 10-10 Feb. 2005.

[18] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *Proc. 31st Int'l Symp. on Microarchitecture (MICRO-31)*, pages 3–13, 1998.

[19] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 128–138, 2000.

[20] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S.-Z. Ueng, J. Stratton, and W. mei Hwu. Program optimization space pruning for a multithreaded gpu. In *Int'l Symp. on Code Generation and Optimization (CGO)*, pages 195–204, April 2008.

[21] Samsung. 512mbit gddr3 sdram, revision 1.5 (part no. k4j52324qc). http://www.samsung.com, June 2006.

[22] Sun Microsystems, Inc. *OpenSPARC^{TM} T2 Core Microarchitecture Specification*, 2007.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture*, pages 24–36, 1995.

[24] L. Zhao, R. Iyer, J. Moses, R. Illikkal, S. Makineni, and D. Newell. Exploring large-scale cmp architectures using manysim. *IEEE Micro*, 27(4):21–33, 2007.