Effect of Instruction Fetch and Memory Scheduling on GPU Performance

Nagesh B. Lakshminarayana Hyesoon Kim
School of Computer Science
Georgia Institute of Technology
{nagesh.bl, hyesoon.kim}@gatech.edu

Abstract

GPUs are massively multithreaded architectures designed to exploit data level parallelism in applications. Instruction fetch and memory system are two key components in the design of a GPU. In this paper we study the effect of fetch policy and memory system on the performance of a GPU kernel. We vary the fetch and memory scheduling policies and analyze the performance of GPU kernels.

As part of our analysis we categorize applications as symmetric and asymmetric based on the instruction lengths of warps. Our analysis shows that for symmetric applications, fairness based fetch and DRAM policies can improve performance. However, asymmetric applications require more sophisticated policies.

1. Introduction

Modern GPUs are designed for throughput computing. They take advantage of the data parallelism available in applications by executing a large number of threads in parallel. Modern GPUs support concurrent execution of 1000s of threads, which results in high Gflop/s. To take advantage of this high computation power, using GPUs for compute intensive applications (GPGPU [10]) is very popular.

GPUs contain multiple Multithreaded processors called Streaming Multiprocessors (SMs). The SMs execute the threads in a GPU kernel in Single Instruction Multiple Thread (SIMT) fashion. SIMT is a form of SIMD execution where the same instruction is fetched and executed for a group of threads. Each thread in the group operates on its own data item. In an SM, threads are grouped into units called Warps and are executed together. A Warp is a group of n threads (in the Tesla architecture [8] on which our GPU simulations are based, n is 32) and represents the basic unit of execution in a GPU. Each SM can typically support several warps at the same time. Thus, each SM could be executing 100s of threads in a MT fashion. Whenever threads are being executed together, the question of resource sharing arises. Even in GPUs, resource allocation is an important problem.

In any multithreaded (MT) architecture, one way to implicitly control the resources allocated to each thread is via instruction fetch. The fetch policy allows direct control of progress made by a thread. By fetching more instructions for a thread, the thread can be made to execute faster than other threads in the system. In an SM, since threads are grouped and executed together as warps, fetch policy can be used to control the resource allocated to each warp. Ideally, the fetch policy should be aware of the progress made by each warp assigned to an SM and should assign more resources to warps that are lagging behind. In case of applications with asymmetric threads/warps, the fetch policy should also be aware of how much more a thread has to execute, the program structure and also the nature of the pending instructions.

Since each SM can execute several warps together, during the execution of a kernel each SM could be executing 100s of threads in a MT fashion. These 100s of threads place more severe demands on the memory system than a multi-threaded application executing on a conventional multi-core CPU. GPU applications try to reduce the pressure on the memory system by having high arithmetic intensity, making use of shared memory and by having as many coalesced memory accesses as possible. GPU applications also try to hide memory access latency by having a large number of threads execute concurrently. In spite of these optimizations and mechanisms, the memory system and in particular, the memory (DRAM) scheduling policy is crucial to the performance of a GPU kernel.[1]

In this paper, we study the effect of the two key aspects explained above - instruction fetch and DRAM scheduling. As part of our study, we classify applications into symmetric and asymmetric based on the lengths of the warps in an application. Our analyses yield the results that fairness oriented fetch and DRAM scheduling policies can improve the performance of symmetric applications, while the best mechanism for asymmetric applications varies from application to application.

2. Background on GPUs

In this section, we describe our baseline GPU architecture, especially instruction fetch and the memory system. Our baseline is based on NVIDIA Tesla architecture used in the GeForce 8-series GPUs [2, 8, 11]. We also compare GPUs with more familiar Multithreaded (MT) or Simultaneous Multithreaded (SMT) processors. Our programming model is based on the CUDA [14] programming language.

2.1. Baseline GPU Architecture

A GPU consists of several Streaming Multiprocessors (SMs) which are connected to a common memory module called the device memory. Modern GPUs such as NVIDIA's GTX280 [7] typically have about 8-32 SMs. Each SM, usually consists of about 8 Scalar Processors (SPs). Each SP is a light-weight in-order processor and shares a common instruction fetch unit with other SPs in the same SM. The instruction fetch unit fetches a common, single instruction that will be executed by all the SPs at the same time. This forms the basis of SIMT execution. Each SM also has a cache called Shared Memory that is shared all SPs in the SM. The Shared Memory provides fast access to data that is shared between threads of the same block.

[1]the terms kernel, benchmark and application will be used interchangeably

Global data and data local to each thread is uncached and is stored in the device memory.

2.2. Instruction Fetch

In an SM, only one instruction is fetched every fetch cycle for a warp selected in a round robin fashion from among the warps currently running on the SM. Warps that are blocked at a barrier, are waiting for loads/stores to complete, or are waiting for a branch to be resolved are not considered for fetching. All fetched instructions are placed in a common issue queue from where they are dispatched for execution. Most instructions takes 4 cycles, some instructions take 8 cycles, and a few take more than 8 cycles.

In spite of supporting multiple hardware threads, all of MT [4], SMT [19] and SM employ a single fetch unit. However, the behavior of the fetch unit is different in each architecture. In MT, multiple instructions are fetched for the same thread in each cycle. In SMT, multiple instructions are fetched for multiple threads in the same cycle, and in an SIMT, a single common instruction is fetched for multiple threads. The fetch unit is critical to the performance of each architecture. MT and SMT fetch policies try to fetch "high quality" instructions [18, 13] into the pipeline. High quality instructions could be instructions from threads with least number of unresolved branches, or from threads with least number of instructions in the pipeline. The goals of these policies is to fetch instructions that have high probability of being executed or fetch instructions from threads that will not clog the pipeline. Their bottomline is to share the pipeline resources between different hardware threads as efficiently as possible. Fetch in SMs is different, since each thread gets its own pipeline, the fetch policy does not have to worry about sharing of functional units or other pipeline structures between the threads. The fetch policy tries to increase throughput by fetching from a different ready warp every fetch cycle.

2.3. Memory System

Each SM has access to a hierarchy of memories. In addition to the device memory, an SM can access registers, shared memory, constant cache and texture cache. All except the device memory, are located within the SM and thus have very short access latencies. Access to device memory takes 100s of cycles. Memory requests from a warp are handled together. When memory requests from the threads of a warp are sequential,[2] the memory requests can be combined into fewer transactions. These kind of accesses are called coalesced memory accesses. However, if the memory addresses are scattered, each memory request generates a separate transaction, called uncoalesced memory accesses.

2.3.1. Merging of Memory Requests

In addition to coalescing, multiple memory requests can be combined into fewer number of memory transactions at various levels in hardware. Figure 1 shows the levels at which merging of memory requests can happen in a GPU. Requests from different SPs inside an SM are held in the Memory Request Queue in the SM. Subsequent requests that overlap with requests already present in the Memory Request Queue are merged with existing requests. This represents *intra-core merging*. Requests from different SMs are accumulated in the Memory Request Buffer in the DRAM controller. If a core makes a request that overlaps with a request already in the Memory Request Buffer, then the new request is merged with the old request. This is an example of *inter-core merging*.

[2]The CUDA manual [15] provides detailed algorithms to identify types of coalesced/uncoalesced memory accesses

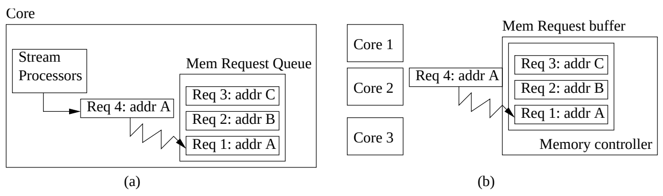

Figure 1. Request mering (a) intra-core merging (b) inter-core merging

We evaluate several fetch and memory scheduling policies, below is a brief description of the evaluated policies. All oracle based polices (ALL, BAR, MEM_BAR, FRFAIR and REMINST) are new polices that we introduce in this study.

2.4. Fetch Policies

1. Round Robin (RR) - RR is the fetch policy that is used in current GPUs. Instructions are fetched for warps in a round robin manner. When sufficient number of blocks are assigned to an SM, RR ensures that SMs are kept busy without wasting cycles waiting for memory accesses to complete.
2. ICOUNT - This is the ICOUNT policy proposed by Tullsen et al. [18] for SMT processors. For each warp, the number of fetched, but undispatched instructions is tracked. In the next fetch cycle, an instruction is fetched for the warp with the fewest number of undispatched instructions. ICOUNT policy tries to increase the throughput of the system by giving higher priority to faster executing threads.
3. Least Recently Fetched (LRF) - In the next fetch cycle, LRF policy fetches an instruction for the warp for which instructions have not been fetched for the longest time. LRF policy tries to ensure that starvation of warps does not occur.
4. Fair (FAIR) - Fair ensures that instructions are fetched for warps in a strictly fair manner. In the next fetch cycle an instruction is fetched for the warp for which the minimum number of instructions have been fetched. This policy ensures that all warps progress in a uniform manner, this increases the probability of merging of memory requests if warps are accessing overlapping regions in memory or regions that are close to each other.
5. Oracle-All (ALL) - This is the first of the three oracle based fetch policies used. In the next fetch cycle, the Oracle-All policy fetches an instruction for the warp which has the most instructions remaining for its completion. Oracle-All gives higher priority to longer warps

and tries to ensure that all warps finish at the same time. When the application is symmetric, Oracle-All behaves similar to FAIR. Figure 2 shows the kernel structures for which the different oracle based policies are designed. Longer columns represent longer warps. For the warps shown in Figure 2(a), Oracle-All prioritizes warp-0 over other warps since it is the longest warp.

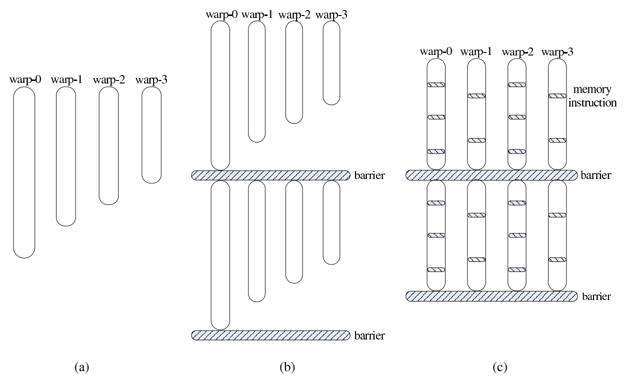

Figure 2. Kernel structures for which different Fetch policies are designed

6. Oracle-Bar (BAR) - The Oracle-Bar policy fetches an instruction for the warp which has the most instructions remaining to its next barrier. If all barriers have been crossed or if the application has no barriers, then the instructions remaining for the completion of the warp are considered. The idea of BAR policy is giving more resources to warps that have the most work before the next barrier. This could reduce the average time for which warps have to wait at barriers for other warps to reach the barrier and thus improve the performance. The BAR policy is designed for applications with warps as shown in Figure 2(b). Since warp-0 has to execute more than the other warps to reach the barrier, BAR gives it higher priority over other warps.

7. Oracle-Mem_Bar (MEM_BAR) - The Oracle-Mem_Bar policy is similar to the Oracle-Bar policy but gives higher priority to warps which have more memory instructions remaining to the next barrier (or completion). If warps have the same number of memory instructions then MEM_BAR behaves identical to BAR. The idea behind MEM_BAR is similar to that of the BAR policy, to reduce the waiting time of warps at barriers. In Figure 2(c), MEM_BAR prioritizes warp-0 and warp-2 since they have more memory instructions until the next barrier compared to warp-1 and warp-3.

2.5. DRAM Scheduling policies

1. First-Come First-Serve (FCFS) - Memory requests are served in the order they arrive, in a First-Come First-Serve manner.

2. First-Ready First-Come First-Serve [16] (FRFCFS) - FRFCFS gives higher priority to row buffer hits. In case of multiple potential row buffer hits (or no row buffer hits), requests are served in a First-Come First-Serve manner.

3. First-Ready Fair (FRFAIR) - FRFAIR is similar to FRFCFS, except that in case of multiple potential row buffer hits (or no row buffer hits), request from the warp which has retired the fewest instructions is served. FRFAIR tries to ensure uniform progress of warps while continuing to take advantage of row buffer hits. This policy is more suitable for symmetric applications than asymmetric applications.

4. First-Ready Remaining Instructions (REMINST) - Like FRFCFS and FRFAIR, REMINST also gives higher priority to row buffer hits. But, in the case of multiple potential row buffer hits (or no row buffer hits), request from the warp which has the most instructions remaining to its completion is served. REMINST gives priority to warps which have longer execution remaining, this ensures that all warps finish at the same time.

Many of the fetch and DRAM scheduling policies explained above try to either ensure that all warps progress uniformly or all warps terminate at the same time. The benefits of doing this are several. This ensures that the occupancy of the SM remains high and also ensures that when one warp blocks (on a load, for example), other ready warps from which instructions can be scheduled are available. Also, if threads are accessing overlapping or neighboring memory regions, this increases the chances of both intra-core and inter-core merging of memory requests.

3. Experimental Setup

3.1. Simulator

We use an in-house cycle accurate, trace-driven simulator for our simulations. The inputs to the simulator are traces of GPU applications generated using the recently released GPUOcelot [12], a binary translator framework for PTX. The GPUOcelot framework provides libraries for emulation of GPU (CUDA) applications. It also allows a trace generator to be attached to GPU applications when they are emulated. We generate per-warp traces instead of per-thread traces since in a GPU, threads are scheduled at warp granularity and not at block/thread granularity. Any divergence exhibited by the threads in a warp is captured in the generated traces.

Table 1 shows the baseline processor/SM configuration used in our simulations.

3.2. Benchmarks

Table 2 shows the benchmarks used in our evaluations and the characteristics of the benchmarks. The first column contains the benchmark names with the kernel names included in parenthesis. Also included in the table are the average and standard deviation of the number of instructions in each warp (avg/std dev of inst.) and the ratio of the length of the longest warp in the kernel to the length of the shortest warp (max/min) in the kernel. Arithmetic intensity shown in the table is the number of non-memory instructions per memory instruction. We use benchmarks from the CUDA SDK 2.2 [1] and the Parboil [3] and Rodinia [6] benchmark suites in our experiments.

Table 1. Baseline processor configuration

Number of cores	8
Front End	Fetch width : 1 instruction, 1KB I-cache, stall on branch, 5 cycle decode
Instruction Fetch	RR (base)
Execution Core	8-wide SIMD execution unit, in-order scheduling all instructions have 4-cycle latency except for FP-DIV (32-cycle), FP-Mul (8-cycle)
On-chip Caches	64KB software manged cache (1 cycle latency), 8 load accesses per cycle 1-cycle latency constant cache, 8 load accesses per cycle 1-cycle latency texture cache, 8 load accesses per cycle

Buses and Memory	2KB row buffer size; DRAM open/close/conflict=70/100/140 cpu cycle; 8 DRAM banks;
	32B-wide, split-transaction core-to-memory bus at 2.5:1 frequency ratio;
	maximum 128 outstanding misses per bank;
	bank conflicts, bandwidth, and queuing delays faithfully modeled

Memory scheduling	FRFCFS (base)

Classification based on Warp length: Based on instruction count of the individual warps, benchmarks are classified as either symmetric or asymmetric. Benchmarks with a max-warp length to min-warp length ratio of about 1.02 i.e., 2% divergence, are classified as symmetric. The category of each benchmark based on this classification is shown in Table 2.

4. Results

In this section we simulate our benchmarks using different fetch and DRAM scheduling policies. In our evaluations, we group applications into symmetric and asymmetric based on data in Table 2. This helps us understand the nature of different types of applications. Within each group of applications, we roughly order applications in the increasing order of memory boundedness.

For all experiments, unless otherwise stated, the combination of RR fetch + FRFCFS DRAM scheduling is used as the baseline. The execution time is the total execution time of a kernel (i.e. the time taken to execute all blocks of a kernel using all cores). Section 4.2 onwards we present data using FRFCFS DRAM scheduling only.

4.1. Effect of Different Fetch Policies

4.1.1. Symmetric Applications Figure 3 shows the performance of symmetric applications using different fetch policies with the FRFCFS DRAM scheduling policy. From the figure we see that all policies other than ICOUNT provide a performance improvement of about 3-6% on average. Only 3 benchmarks - SimpleGL, BlackScholes, MonteCarlo - show significant performance improvement. Since all benchmarks are symmetric, even ALL makes threads progress uniformly, similar to FAIR and also gives performance similar to FAIR. For applications without barriers (only NBody, MatrixMul and MonteCarlo have barriers among symmetric applications), BAR is identical to ALL. In general, since all benchmarks are symmetric, in addition to FAIR, even ALL, BAR, MEM_BAR make threads progress uniformly.

The experiment for Figure 4 is identical to the experiment for Figure 3, except that Figure 4 uses FCFS for DRAM scheduling instead of FRFCFS. On average, none of the combinations using FCFS provide any performance improvement over FRFCFS based policies. Most of the memory intensive benchmarks (BlackScholes onwards, from left to right) show performance degradation with FCFS. It is because FRFCFS performs significantly better than FCFS and none of the fetch polices can offset the performance degradation due to FCFS.

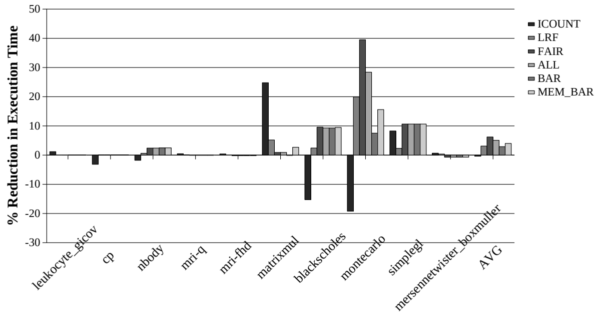

Figure 3. Performance of Symmetric applications with different fetch policies and FRFCFS DRAM scheduling

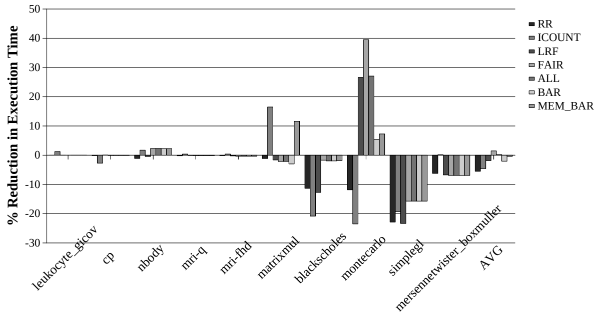

Figure 4. Performance of Symmetric applications with different fetch policies and FCFS DRAM scheduling (baseline: RR + FRFCFS)

Results for experiments with FRFAIR and REMINST are shown in Figure 6 and Figure 7, respectively. Compute intensive benchmarks (GICOV_kernel from Leukocyte, cp, mri-q, mri-fhd) do not show any performance improvement even with different DRAM policies. However, memory intensive benchmarks show considerable improvement with FRFAIR and REMINST policies. These policies like FRFCFS give priority to row buffer hits. In addition, FRFAIR tries to ensure that threads progress in a uniform manner. This tends to further improve performance of symmetric applications by increasing merging (discussed shortly) and row buffer hits. Since the applications are symmmetric, REMINST behaves similar to FRFAIR and provides similar benefits.

Next, let's look at why some applications show performance improvement for certain policy combinations. Sim-

Table 2. Benchmarks Characteristics(Arith. Int: Arithmetic Intensity)

Name	origin	# blocks x #warps per block	avg/std dev of inst.	max/min	sym?	Arith. Int.
MonteCarlo (MonteCarloOneBlockPerOption)	SDK 2.2	256 x 8	15468 / 24	1.005	symmetric	14.1
MersenneTwister (BoxMullerGPU)	SDK 2.2	32 x 4	58638 / 0	1.000	symmetric	4.0
Nbody (integrateBodies)	SDK 2.2	32 x 8	153910 / 0	1.000	symmetric	3460.1
BlackScholes (BlackScholesGPU)	SDK 2.2	480 x 4	6348 / 30	1.015	symmetric	18.1
MatrixMul (matrixMul)	SDK 2.2	40 x 8	299 / 0	1.000	symmetric	19.3
BinomialOptions (binomialOptionsKernel)	SDK 2.2	64 x 8	161192 / 5835	1.109	asymmetric	273.5
ConvolutionSeparable (convolutionColumnGPU)	SDK 2.2	96 x 4	475 / 72	1.618	asymmetric	16.8
Dxtc (compress)	SDK 2.2	64 x 2	10042 / 1330	1.333	asymmetric	287.1
PostProcessGL (cudaProcess)	SDK 2.2	1024 x 8	889 / 38	1.206	asymmetric	31.6
ScalarProd (scalarProdGPU)	SDK 2.2	128 x 8	869 / 105	1.405	asymmetric	12.1
DwtHaar1D (dwtHaar1D)	SDK 2.2	256 x 16	393 / 280	6.348	asymmetric	22.6
Histogram256 (mergeHistogram256Kernel)	SDK 2.2	256 x 2	130 / 30	1.590	asymmetric	42.0
Eigenvalues (bisectKernelLarge_MultInterval)	SDK 2.2	23 x 8	535383 / 345873	183.223	asymmetric	931.8
SimpleGL (kernel)	SDK 2.2	1024 x 2	74 / 0	1.000	symmetric	7.6
VolumeRender (d_render)	SDK 2.2	1024 x 8	2642 / 3491	80.743	asymmetric	694.0
mri-fhd (ComputeFH_GPU)	Parboil	128 x 8	12073 / 0	1.000	symmetric	861.4
mri-q (ComputeQ_GPU)	Parboil	128 x 8	18983 / 0	1.000	symmetric	1354.9
cp (cenergy)	Parboil	256 x 4	168087 / 0	1.000	symmetric	8844.8
pns (PetrinetKernel)	Parboil	18 x 8	338250 / 3909	1.037	asymmetric	20.5
leukocyte (GICOV_kernel)	Rodinia	596 x 6	52755 / 5	1.001	symmetric	17585.0
leukocyte (dilate_kernel)	Rodinia	796 x 6	19528 / 533	1.334	asymmetric	3252.5
leukocyte (MGVF_kernel)	Rodinia	36 x 10	193 / 26	1.489	asymmetric	26.6
cell (evolve)	Rodinia	216 x 16	970 / 632	10.490	asymmetric	45.9
needle_gpu (needle_cuda_shared)	Rodinia	63 x 1	2503 / 34	1.036	asymmetric	17.9

pleGL and BlackScholes show improvement with FAIR and the Oracle based mechanisms due to the increase in MLP exposed by these mechanisms and also due to increase in DRAM row buffer hit ratio due to these mechanisms as shown in Figures 16 and 20.

ICOUNT policy which performs well on SMT machines, results in peformance degradation for most applications. The vanilla version of ICOUNT does not work well when there are several threads in the system. ICOUNT repeatedly keeps fetching for only a subset of the threads in the system while starving others. This happens because ICOUNT always examines threads in a particular order and if threads at the beginning of the order have fewer pending or the same number of pending instructions as other threads, instructions are fetched for the threads at the beginning of the order. If ICOUNT is coupled together with something like a round-robin mechanism, it could provide benefit for a greater number of applications. MatrixMul is one of the few applications which shows benefit with ICOUNT. MatrixMul is a small kernel with a few memory operations and barriers. All policies other than ICOUNT cause warps to progress uniformly and warps reach barriers or execute memory operations at roughly the same time. Because of this there are a lot of idle cycles. Since threads are progressing uniformly, I-Cache misses stall all threads as well. However, in the case of ICOUNT policy, the threads diverge considerably, and thus, even if there is a barrier or an I-Cache miss, some threads will be ready to execute resulting in reduced idle cyles and improved performance.

Some of the policy combinations provide benefit for MonteCarlo which is a memory intensive benchmark. Investigations into the performance improvement of MonteCarlo show that the improvement is mostly due to the merging of memory requests in the memory request queue of each SM (Section 2.3.1). In MonteCarlo, threads in a block access consecutive memory locations and threads with matching indices in each block access the same memory locations. Requests from threads in the same warp are coalesced and reach the memory request queue. If requests from different warps reach the memory request queue while the queue contains requests from other warps to the same or neighboring locations, then the requests from different warps can be merged. Figure 5 shows the portion of code from MonteCarlo that causes merging of memory requests. In line 6, threads with matching indices across different blocks access the same memory address (load from array d_Samples). Thus the memory requests generated after coalescing requests from individual threads can be merged since they can be satisfied by a single request. Thus MonteCarlo has a lot of potential for merging of requests in an SM. This potential is realized when warps across blocks make progress in step with each other. Fetch policies such as FAIR, ALL which cause warps in symmetric applications to proceed in-step with each other provide considerable benefit over the default RR fetch which can cause warps to diverge from each other. This analysis is in agreement with Figure 18 which shows the number of merged memory requests for symmetric application. We discuss the merging effect in Section 4.4. FAIR fetch policy provides the greatest performance improvement for MonteCarlo by causing more merges than other fetch policies. Since FAIR strictly ensures that warps progress uniformly, requests from different warps to the same memory location will reach the memory request queue at approximately the same time with high probability, this increases the number of merges.

Benchmark	Reason
BlackScholes	MLP + DRAM row hits
MonteCarlo	Memory request merging
SimpleGL	MLP + DRAM row hits
MersenneTwister	DRAM row hits

Table 3. Reasons why Symmetric benchmarks show benefit with different policy combinations

MersenneTwister which is also a memory intensive ap-

```
...
for(iSum = threadIdx.x; iSum < SUM_N; iSum += blockDim.x)
{
  __TOptionValue sumCall = {0, 0};
  for(int i = iSum; i < pathN; i += SUM_N)
  {
    real r = d_Samples[i];
    real callValue = endCallValue(S, X, r, MuByT,
                                  VBySqrtT);
    sumCall.Expected += callValue;
    sumCall.Confidence += callValue * callValue;
  }
  s_SumCall[iSum]  = sumCall.Expected;
  s_Sum2Call[iSum] = sumCall.Confidence;
}
...
```

Figure 5. MonteCarlo

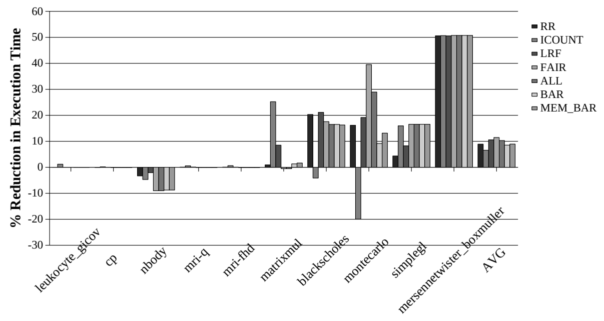

Figure 6. Performance of Symmetric applications with different fetch policies and FRFAIR DRAM scheduling

plication shows considerable improvement with FRFAIR and REMINST DRAM policies due to increase in row buffer hit ratio with these policies.

4.1.2. Asymmetric Applications

Figures 8, 9 and 10 show the performance of asymmetric applications with different fetch and DRAM scheduling policies. On average, none of the policy combinations provide performance improvements over the default combination of RR fetch and FRFCFS DRAM scheduling.

Some of the compute intensive benchmarks such as dilate_kernel (Leukocyte), Eigenvalues, BinomialOptions do not show much variation with change in fetch and DRAM policies. Eigenvalues shows slight decrease in performance with FAIR and ALL fetch policies due to increase in waiting time at barriers. Cell, which is a compute intensive benchmark, shows decrease in performance for all fetch policies except LRF. Cell is a very asymmetric benchmark. In each block, some warps are very short compared to the rest of the warps. The max by min value in Table 2 shows this. Due to the highly asymmetric nature of the warps, the oracle based policies prioritize the longer warps over the shorter warps resulting in very few instructions being fetched for the shorter warps. For Cell, this increases the number of idle cycles since the number of active threads is effectively reduced. Since FAIR tries to enforce fairness, it fails due to the asymmetric nature of warps and results in bad performance.

Some memory intensive benchmarks such as ConvolutionSeparable, Histogram256, ScalarProd and pns show considerable performance improvement with certain combinations of fetch and DRAM policies.

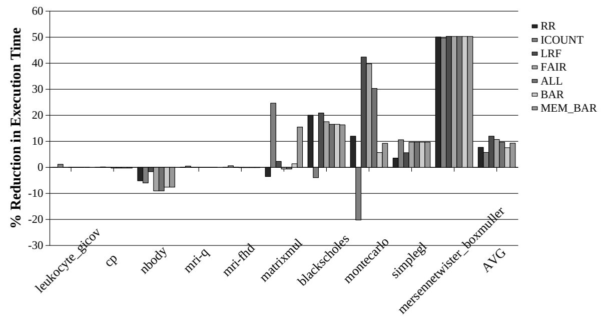

Figure 7. Performance of Symmetric applications with different fetch policies and REMINST DRAM scheduling

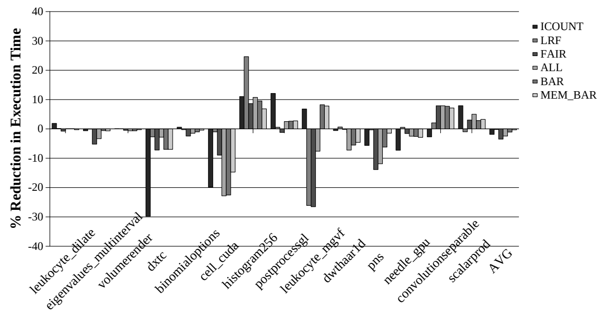

Figure 8. Performance of Asymmetric applications with different fetch policies and FRFCFS DRAM scheduling

Though ConvolutionSeparable is classified as an asymmetric application, a large number of warps (blocks) in it are identical. Only the blocks operating on boundary pixels of the image (Convolution is an image processing technique) are different from the rest of the blocks which are identical to each other. The blocks operating on the boundary pixels are also identical to each other. Thus ConvolutionSeparable consists of two groups of identical blocks and tends to behave similar to symmetric applications showing good performance with fairness oriented mechanisms.

Each warp in Histogram256 is short and has uncoalesced memory accesses (i.e., 32 memory requests) at the beginning of each warp. Compared to other policies, RR is slow to respond to completion of I-Cache and Load misses. What we mean by slow to respond is, when an I-Cache miss or a load miss is satisfied for a warp, because of the way RR works, there is a certain delay before RR fetches again for that warp. Other policies, because of their design respond faster to I-cache miss or load miss completions. This is one of the key reasons why RR performs worse compared to other fetch policies. Other factors such as amount of merging also affect the performance.

Dxtc and Dwthaar1D are asymmetric applications with few barriers. In addition to the difference in overall lengths, there is a considerable variation in number of instructions between consecutive barriers for different warps. Because of this structure, many of the policies perform badly. FAIR tries to ensure uniform progress and fails. ALL fetches only for the longest threads, while BAR (or MEM_BAR) fetches only for thread

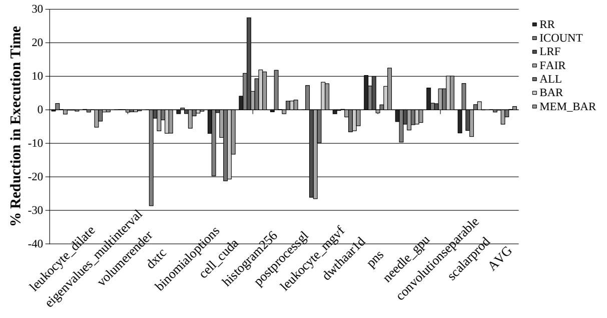

Figure 9. Performance of Asymmetric applications with different fetch policies and FRFAIR DRAM scheduling

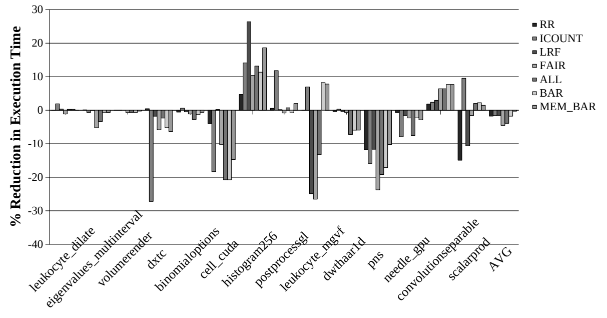

Figure 10. Performance of Asymmetric applications with different fetch policies and REMINST DRAM scheduling

with the most instructions (or memory instructions) to the next barrier.

Needle, another memory intensive benchmark, slows down with ICOUNT due to reduction in MLP compared with other fetch policies.

We plan to investigate in detail the performance of IMGVF_kernel (Leukocyte), pns, ScalarProd and other interesting results that are not discussed here to improve our understanding of the behavior of GPU applications.

Observations: From the data for asymmetric applications it is clear that no particular combination of fetch and DRAM policy can provide good benefit. The fetch or DRAM policy or the policy combination that can provide benefit is dependent on the application. To come up with a single policy/policy combination that can improve the performance of asymmetric applications we have to study the structure and behavior of applications more thoroughly.

4.2. Barrier Wait Time of a warp

We measure the average wait time at a barrier for a warp for both symmetric and asymmetric applications. Intuitively, the performance of applications would improve if the average barrier wait is reduced. The BAR and MEM_BAR fetch policies were designed with this goal of reducing average barrier wait time. In Figures 11 and 12, we show the total barrier wait time of a warp which is computed from the average wait time.

4.2.1. Symmetric Benchmarks

Figure 11 shows the total of the time spent by a warp at all barriers, waiting for other

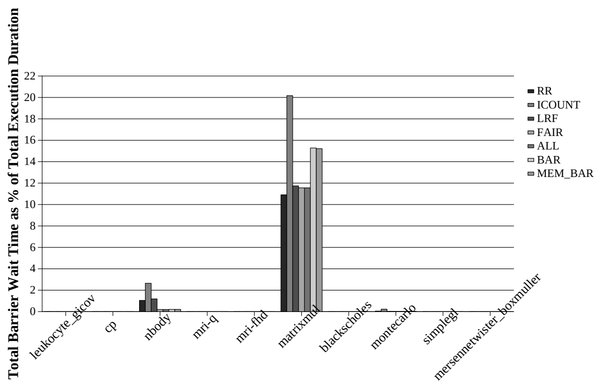

Figure 11. Total barrier wait time for a warp (average barrier wait time x number of barriers x 100 / total execution cycles) for Symmetric applications

warps of the same block as a percentage of the total execution duration of the application. Symmetric applications typically contain very few or no barriers at all, since most of the warps tend to be identical. Among symmetric applications, only MatrixMul has significant barrier wait time. Though ICOUNT has the longest wait time at a barrier for matrix multiplication, it provides the best performance as well. This is in accordance with what was explained earlier. ICOUNT causes threads to diverge considerably compared to other policies and hence has longer barrier wait time. This is the same reason why ICOUNT has longer wait time for NBody also.

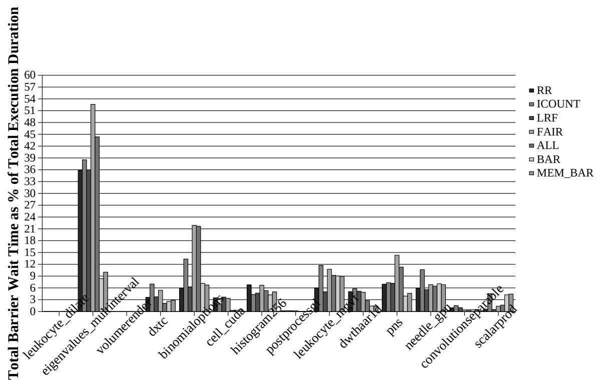

Figure 12. Total barrier wait time for a warp (average barrier wait time x number of barriers x 100 / total execution cycles) for Asymmetric applications

4.2.2. Asymmetric Benchmarks

Figure 12 shows the total time spent by a warp at all barriers for asymmetric applications. Two of the asymmetric kernels - dilate_kernel (Leukocyte) and VolumeRender - do not contain barriers at all. For many kernels with barriers, BAR and MEM_BAR have the lowest wait times. For some applications, reduction in wait times does result in better performance and for others reduction in wait times does not result in performance improvement.

For pns, FAIR causes the longest average barrier wait time and for Cell, it is FAIR along with LRF. However, for other benchmarks such as ScalarProd, the fetch policy that causes the longest average barrier wait time is not the same as the worst performing fetch policy. For ScalarProd, the worst performing fetch policy is RR while the fetch policies causing the longest average wait are BAR and MEM_BAR. Though larger barrier wait times increase execution duration, a direct inference about which mechanism would perform worst cannot be made based on the barrier wait time with each mechanism.

4.3. Average Latency of Memory Operation

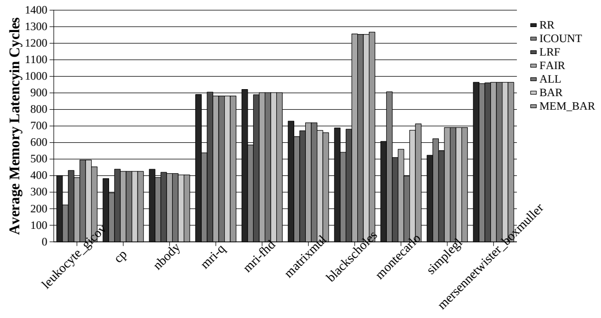

Figure 13. Average latency of memory instructions for Symmetric applications

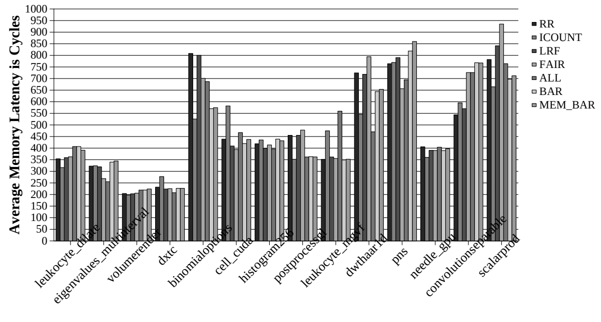

Figure 14. Average latency of memory instructions for Asymmetric applications

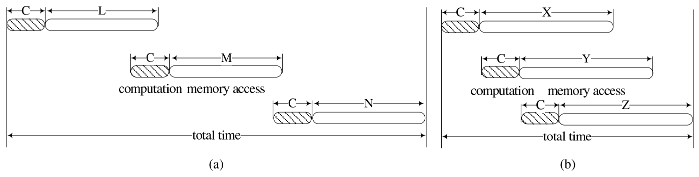

Figure 15. Memory latency for different cases : (a) MLP = 1, (b) MLP > 1

The average latency of a memory access is shown in Figures 13 and 14. This is the duration taken by a memory operation to complete once a memory request has been dispatched by the SM. We see that some of the compute intensive benchmarks such as GICOV_kernel (Leukocyte), NBody and cp have low average memory latencies. Others such as mri-q, mri-fhd and MatrixMul have high latency per memory operation though they have few memory accesses per warp. This is because the memory accesses in these kernels are concentrated in small regions of code, which causes the average latency of memory operations to increase since the number of running warps per SM is fairly high (32 warps per SM). The MLP values (Figure 16) for these applications confirm this. Memory intensive benchmarks such as BlackScholes and MonteCarlo, naturally have large number of memory requests and have longer memory operation latency.

For memory intensive benchmarks, higher average memory latency does not necessarily mean bad performance. The Fetch policy with the highest MLP (policy that causes most memory operations to be issued close to each other across all warps executing on all SMs) often has the worst average latency. Since memory requests can be delayed by other requests, the latency of memory operations goes up naturally. However, memory requests are processed in parallel by the DRAM controller, this reduces the total contribution of memory instructions to the total execution duration and improves performance.

Besides MLP, the average latency is dependant on another factor - the amount of merging. If the MLP is low, then the average latency is also going to be low. On the other hand, if the MLP is high, the average latency is also usually high. However, if merging of requests is considerable, then the average latency is going to be lower.

Figure 16 shows the MLP for symmetric applications. We define instantaneous MLP of the application as the number memory requests that are active when a new memory request is issued. We average the instantaneous MLP for each memory request to obtain the average MLP for the application. GPU architecture naturally has high MLP due to high number of concurrently running threads. Hence, unlike in a single thread processor, high MLP does not result in better performance. High MLP usually means memory requests are bursty in nature.

Figure 15 shows how MLP can affect the average latency of memory instructions. In Figure 15(a), where the MLP is equal to one, the memory instruction from each warp does not experience contention from other memory instructions and thus each can complete in L, M and N cycles. In Figure 15(b), the memory instruction from each warp has to contend with memory instructions from other warps. There might be contention for the DRAM bus and channel and bank conflicts as well. Presence of other requests in the DRAM scheduler also delays the scheduling of a memory request. Thus in this case, each memory request takes longer to complete and they complete in X, Y and Z cycles, each of which is going to be longer than any of L, M and N (assuming no row buffer hits).

4.4. Number of Merges

Figures 18 and 19 show the ratio of memory requests merged to the number of memory requests. We only show intra-core merging because inter-core merging is very small.

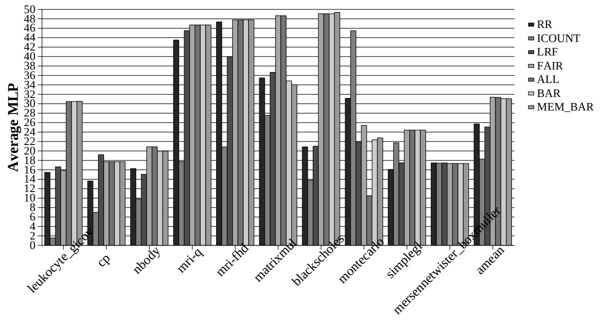

Figure 16. MLP for Symmetric applications

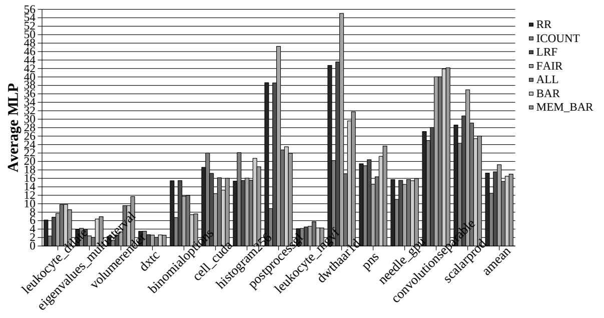

Figure 17. MLP for Asymmetric applications

Merging of requests reduces the number of requests sent out to the device memory and thus can help speed up applications significantly. All applications except MersenneTwister and BinomialOptions, can derive the benefit of merging. Due to merging, the execution duration of MonteCarlo with FAIR and FRFCFS is reduced by 40% when compared to execution with RR and FRFCFS.

For merging to happen, threads across warps of blocks assigned to the same SM should access memory addresses that are close to each other. In such a situation, two memory requests could be merged if they could be satisfied by a single memory transaction.

For symmetric applications, enforcing fairness ensures that memory requests by the same static instruction across different warps reaches the memory request queue at about the same time. If multiple requests can be satisfied by a single transaction, then the requests are merged. In many GPU applications, the code is such that the same static instruction across different warps accessses neighboring or overlapping addresses. Thus, by ensuring uniform progress, the amount of merging and hence the performance can be improved.

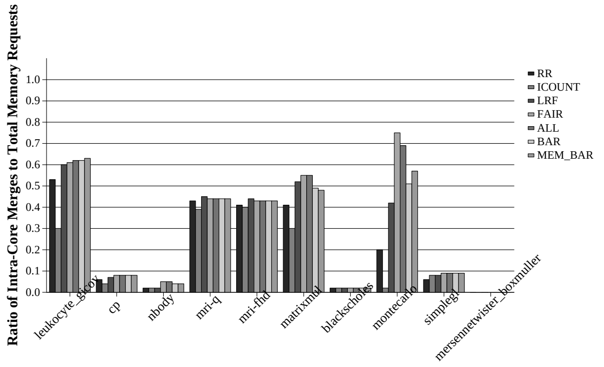

Figure 18. Ratio of intra-core merges for Symmetric applications

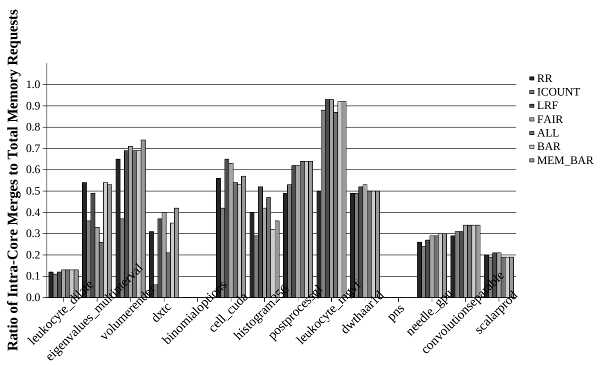

Figure 19. Ratio of intra-core merges for Asymmetric applications

4.5. DRAM Row Buffer Hit Ratio

Figures 20 and 21 show the DRAM row buffer hit ratios for symmetric and asymmetric applications. Intuitively, for symmetric applications, fairness should improve the DRAM row buffer hit ratio and improve the performance. However, though fairness increases the hit ratio, especially for memory intensive, symmetric applications, it does not result in performance improvement always.

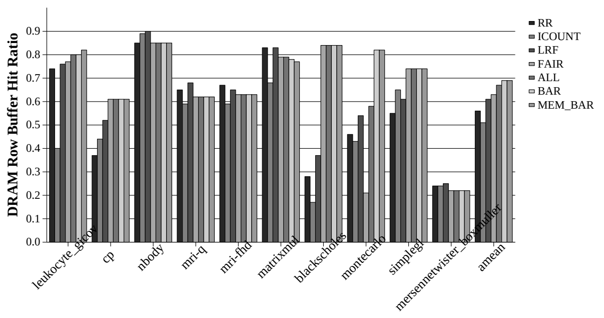

Figure 20. DRAM row buffer hit ratio for Symmetric applications

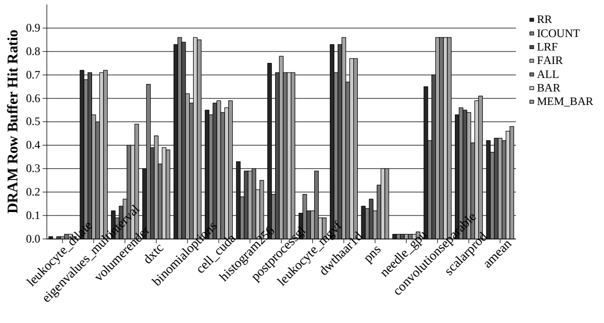

Figure 21. DRAM row buffer hit ratio for Asymmetric applications

5. Related Work

Our paper evaluates the performance of different fetch and DRAM scheduling policies on GPU kernels. To the best of our knowledge, this is the first work that does the evaluation of fetch and DRAM policies for GPUs. There have been several papers in the past that have evaluated fetch and DRAM scheduling for SMTs and other architectures. Here we present SMT-based fetch polices and discuss why those polices would necessarily not provide benefit in the GPU architectures.

In [18], Tullsen et al., propose several fetch policies such as BRCOUNT, MISSCOUNT, ICOUNT. These policies improve throughput by improving the quality of the instructions that are fetched. BRCOUNT fetches instructions from threads that have the least unresolved branches in the pipeline. This ensures that the fetched instructions are less likely to be flushed. ICOUNT fetches instructions from threads that have the least number of threads in the pipeline. ICOUNT policy favors fast executing threads. We evaluated the ICOUNT policy in our paper and found that it performs badly for GPUs. BRCOUNT and MISSCOUNT cannot be applied to a GPU since we do not fetch instructions for a warp when we detect either a pending branch or a load miss for that warp.

Luo et al. [13] evaluate several SMT policies aiming to provide both throughput and fairness to the threads in the system. The evaluated policies employ fetch prioritization and fetch gating to ensure high throughput and fairness. The evaluated policies include RR, ICOUNT, Low Confidence Branch Prediction Count (LC-BPCOUNT) and variations of these. Though we evaluate fetch prioritization we do not evaluate fetch gating. Fetch gating based on instruction count could have been a possible variation to the evaluated fetch policies, but with the large issue queue and scheduler window employed in GPU architectures we doubt whether fetch gating would have made much of a difference.

In several works in the past ([17] [5]), long latency loads are handled either by fetch stalling or by flushing the thread experiencing the load miss. In our simulations, we fetch-stall warps experiencing load misses. Flushing warps with load missed might not be very beneficial since those warps will be occupying only a small fraction of the front-end resources due to presence of a large number of warps in the SM.

Eyerman et al. [9] suggest 1) fetch-stalling or flushing threads on a load miss if they have no MLP, and 2) allocate more resources to threads on a load miss if they have MLP. Allocating more resources to the same warp to expose MLP might provide minor benefits for some applications, but because most applications are symmetric in the GPU kernels (i.e. all thread will have the same or similar MLP), this policy will not be applicable.

6. Conclusion

In this paper, we try to understand the effect of fetch and DRAM scheduling policies on the performance of GPU kernels. We run GPU kernels with different combinations of fetch and memory scheduling policies and analyze their performance. The different fetch policies are the default round robin policy, a fair fetch policy, the ICOUNT policy [18], least recently fetched policy and also a few oracle based policies. The memory policies we try include: FCFS, FRFCFS and two oracle based variations of FRFCFS.

Our experiments yield the following results: (1) Computation intensive benchmarks are insensitive to fetch and DRAM scheduling polices since the processor can always find warps to execute, (2) For symmetric applications, fetch and DRAM policies which try to provide fairness to warps can improve performance when compared to the default fetch and DRAM policies. (3) For asymmetric applications, no single fetch and DRAM policy combination can provide performance benefit for a good number of the evaluated applications, and (4) merging effect is significant for some applications and fairness based policies increase the amount of merging.

Unfortunately, different applications show benefits with different combinations of fetch and DRAM policies. Simple straight forward mechanisms would not provide benefit for asymmetric applications even with oracle information. Mechanisms that are aware of program structure and behavior would be necessary for asymmetric applications.

In future work, we will further analyze the behavior of asymmetric applications and try to improve their performance by coming up with new fetch and DRAM policies.

References

[1] CUDA SDK.

[2] GeForce GTX280. http://www.nvidia.com/object/geforcefamily.html.

[3] Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.

[4] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: a processor architecture for multiprocessing. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, May 1990.

[5] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Optimizing long-latency-load-aware fetch policies for smt processors. *International Journal of High Performance Computing and Networking*, 2(1):45–54, 2004.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization*, 2009.

[7] N. Corporation. Nvidia products. `http://www.nvidia.com/page/products.html`.

[8] E. Lindholm, J. Nickolls, S.Oberman and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March-April 2008.

[9] S. Eyerman and L. Eeckhout. A memory-level parallelism aware fetch policy for smt processors. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 240–249, February 2007.

[10] GPGPU. General-Purpose Computation Using Graphics Hardware. http://www.gpgpu.org/.

[11] John Nickolls, Ian Buck, Michael Garland and Kevin Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, March-April 2008.

[12] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of ptx kernels. In *IEEE International Symposium on Workload Characterization*, 2009.

[13] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, pages 164–171, 2001.

[14] NVIDIA Corporation. Cuda (compute unified device architecture). http://www.nvidia.com/cuda.

[15] NVIDIA Corporation. CUDA Programming Guide, April 2009.

[16] 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.

[17] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 318–327, 2001.

[18] D. M. Tullsen, S. J. Eggers, J. S. Emer, and H. M. Levy. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *ISCA-23*, pages 191–202, 1996.

[19] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA-22*, pages 392–403, 1995.