Performance in GPU Architectures: Potentials and Distances

Ahmad Lashgar
School of Electrical and Computer Engineering
College of Engineering
University of Tehran
a.lashgar@ece.ut.ac.ir

Amirali Baniasadi
Electrical and Computer Engineering Department
University of Victoria
amirali@ece.uvic.ca

Abstract

GPUs can execute up to one TFLOPs at their peak performance. This peak performance, however, is rarely reached as a result of resource underutilization. Three parameters contribute to this inefficiency: branch divergence, memory access delays and limited workload parallelism. To this end we suggest machine models to estimate performance gain potentials obtainable by eliminating each performance degrading parameter. Such estimates indicate how much improvement designers could expect by investing in different GPU subsections. Moreover, our models show how much performance is lost compared to an ideal GPU as a result of non-ideal GPU components. We conclude that memory is by far the most important parameter among the three issues impacting performance. We show that in the presence of an ideal memory system, GPU performance can reach within 59% of an ideal system. Meantime, using an ideal control-flow mechanism or unlimited execution resources does not come with the same impact. In fact, as we show in this study, an ideal control-flow could harm performance as the result of increasing pressure on the memory system.

In addition, we study our models under GPUs exploiting aggressive memory systems and well-equipped Stream Multiprocessors. We investigate how previously suggested control-flow solutions impact performance-degrading issues and make recommendation to enhance control-flow mechanisms.

1. INTRODUCTION

Graphics processing units (GPUs) are used as accelerators for general-purpose throughput-intensive workloads in high performance computing. Modern GPUs are capable of reaching peak processing power of one TFLOPS for single-precision operations and 500 GFLOPS for double-precision operations in just a single chip [1]. In addition to this dense processing capability, GPUs come with little power dissipation providing further motivation for deploying them in computers in order to maximize performance per cost [4].

One of the major concerns of GPU designers is the underutilization of GPU's computational resources during execution of general-purpose applications [2]. General-purpose applications are divided into memory-intensive and computation-intensive classes.

Memory-intensive applications underutilize GPU resources mainly as the result of high pressure on the memory. The pressure can increase under unpredictable memory access patterns [12]. While GPU's typical memory bandwidth is about 5x higher than CPUs [10], GPUs are restricted by the fact that their bandwidth is shared among thousands of threads [14]. This limitation does not appear as serious for graphical workloads since in such workloads, threads share large data sets [3]. Consequently, GPUs are capable of providing the required bandwidth for graphical applications [13].

Computation-intensive applications underutilize GPU resources often as a result of frequent occurrence of branch instructions [12, 15]. In contrast to conventional graphical computations, general-purpose workloads typically include a high number of branch instructions [11]. GPUs execute groups of threads in lock-step. All the threads that belong to the same group (also referred to as *warps*) execute the same instruction but use different data (an SIMD-like model referred to as SIMT by NVIDIA [3]). Lock-step execution is violated if the branch instruction diverges, dividing the warp into two sub-groups with different program counters [7]. This can impact performance negatively. Current GPUs use masking and re-convergence [9] to address this issue.

There have been many studies at different levels to address resource underutilization in GPUs. These studies could be categorized into *memory solutions* and *control-flow mechanism solutions*. Memory solutions aim at either reducing or hiding memory latency. Control-flow solutions reduce the impact of branch divergence by regrouping the threads into new warps.

On-chip last-level shared caches [16] are commonly used to reduce average memory latency. On the off-chip side, DRAM scheduling policy can also impact memory latency [19, 20]. Hiding memory latency generally is effectively feasible under higher number of concurrent threads. In the absence of high concurrency, it is possible to diverge the warp and let the hit-threads (threads that hit in the cache) continue [17], hiding the latency of miss-threads.

Basic control-flow mechanisms divide warps into independently schedulable warps on every diverging branch instructions. Over-divergence occurs when these divided warps experience a diverging branch again. Stack-based reconvergence mechanisms prevent over-divergence of a warp with thread-masking and reconvergence [9]. Other studies have used dynamic warps to keep all threads active and avoid temporal inactivation due to thread-masking. Fung et al. [7] suggested dynamically reconstructing the warps to keep threads of the same diverging path in the same warp. Meng et al. [8] suggested splitting the warp into independently schedulable warp-splits. However, unlike [9] and [7], this mechanism reconverges these warp-splits after execution of the diverging paths and revives the original warp.

Our study shows that greedy control-flow mechanisms aiming at maximizing computational resource utilization harm coalescing memory accesses [3] and increase pressure on off-chip memory significantly.

In addition to memory and branch divergence, the third parameter impacting performance in GPUs is the available Streaming Multiprocessor (SM) resources, which could limit the number of concurrent Cooperative Thread Arrays (CTAs) for highly parallel applications. An increase in such resources would benefit applications that come with enough parallelization.

Finding new and balanced solutions to facilitate better usage of GPUs requires a deep understanding of current obstacles and their impact on previously suggested solutions. Therefore in this work we study the impact of branch divergence, memory and execution resource limitations on GPUs.

In summary we make the following contributions:

- We develop a set of machine models to show how much speedup can be reached when each one of GPUs' performance-impacting parameters is idealized. Moreover, we compare our models to an ideal machine to show how each parameter can impact an otherwise ideal GPU.
- We show that while improving any of the three performance-impacting parameters can enhance performance, it is the memory that has the greatest impact. We also show that an ideal memory system can improve performance and reduce the impact of other parameters.
- We explore abnormal trends in our results. In particular we show how and why a non-ideal control-flow solution can outperform an ideal control-flow mechanism as the result of memory load variations.
- We introduce machine models to investigate how different practical limitations distance us from peak and ideal performance. We show that peak performance is highly sensitive to memory performance and less dependent on per SM resources and control-flow behavior.
- We explore whether our findings remain valid under changes in GPU configurations and possible technology trends.

The rest of this paper is organized as follows. In Section 2 we review background. In Section 3 we discuss our methodology. In Section 4 we present and evaluate the machine models used in this work. In Section 5 we study sensitivity analysis and investigate whether our findings stay valid for alternative GPU configurations.

2. BACKGROUND

GPUs are built of hundreds of processing elements (*PE*). Each cluster of PEs sharing resources is an SM. Scalable arrays of SMs are connected to memory controllers through an interconnection network. As shown in Figure 1, an SM has a *thread pool,* which stores the *status* (no context) of outstanding threads. The pool is shared among CTAs.

A group of threads (referred to as a warp) are executed on an SM concurrently. The *fetch* and *decode* stages of the pipeline in each SM are shared among all threads of the warp. The pipeline's execution bandwidth at *Register Read* and *Execution* stages is the *SIMD width* and depends on the number of PEs per SM. Sharing the simple fetch and decode stages of this in-order pipeline facilitates providing enough chip area for hundreds of PEs.

In conventional GPUs a warp executes the same instruction from different threads concurrently. Meantime threads are allowed to diverge and execute different instructions. Conditional branch instructions, however, can cause the warp to diverge (known as the branch divergence hazard) resulting in inefficient resource utilization. Control-flow mechanisms are required to remove or reduce the cost associated with this hazard. In this work we study GPUs using two

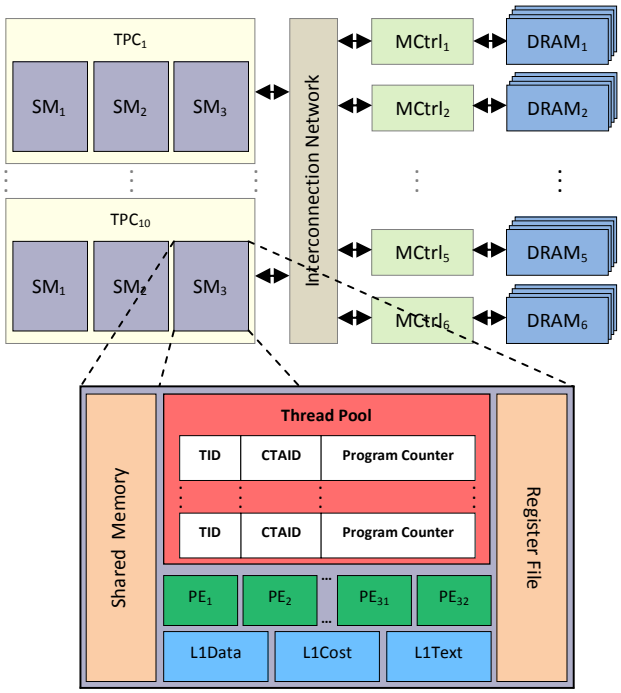

Figure 1. Stream Multiprocessor shares CTA limiting resources (thread pool, register file and shared memory) between concurrent CTAs.

well-known control-flow mechanisms, i.e. postdominator reconvergence (PDOM) and dynamic warp formulation (DWF).

PDOM, which is used in GPUs such as NVIDIA G80[3], relies on *static warp* formulation. Individual threads that construct a warp do not change during threads' lifetime. Placement of the threads in a warp depends on their inter CTA unique id (TID in CUDA terminology [2].) PDOM uses round-robin to issue the next warp to the pipeline. PDOM uses masking to deal with branch divergence. Masking stores multiple program counters (PC) per warp. The mask vector for each PC represents threads associated with the PC. During executing an instruction with a particular PC, the mask temporarily inactivates the threads not associated with the PC. PDOM uses a stack per warp to prevent over-divergence and to guarantee that vector masks associated with the same PC will eventually merge. This is done by using additional information (reconvergence point) passed to branch instruction [7].

PDOM comes with the advantage of guaranteeing to execute top-level basic blocks (starting with reconvergence points) with maximum SIMD utilization. Memory accesses belonging to threads with close TIDs are usually performed in the same SM. PDOM benefits from this locality as it places close TIDs in the same warp and coalesces their memory accesses whenever possible. Meantime, PDOM shows poor performance in the presence of large number of instructions in divergent paths. Another PDOM disadvantage is due to the fact that masking and inactivating threads results in avoiding execution of threads that may be ready and if executed can potentially hide the latency of memory accesses.

The idea behind DWF is to merge the warps with the same divergent path to maintain high utilization under SIMD. DWF cannot reach this goal unless enough warps with the same divergent path exist.

Since a thread may be scheduled in different warps under DWF, DWF uses lane aware scheduling and preserves SIMD lane of the thread in any warp. Lane aware scheduling uses the original

Table 1. Baseline configurations.

Parameter	Value
NoC	
Total Number of SMs	30
Number of Memory Ctrls	6
Number of SM Sharing an Interconnect	3
SM	
Warp Size	32 Threads
Number of Thread per SM	1024
Number of Register per SM	16384 32-bit
Number of PEs per SM	32
Shared Memory Size	16KB
L1 Data Cache	64KB:8-way:64BytePerBlock
L1 Texture Cache	8KB:2-way:64BytePerBlock
L1 Constant Cache	8KB:2-way:64BytePerBlock
Clocking	
Core Clock	325 MHz
Interconnect Clock	650 MHz
DRAM memory Clock	800MHz
Memory	
Number of Banks Per Memory Ctrls	8
DRAM Scheduling Queue Size	32
DRAM Scheduling Policy	Fast Ideal Scheduler
GDDR3 Memory Timing	tRRD=12, tRCD=21, tRAS=13, tRP=34, tRC=9, tCL=10
Control-Flow Mechanisms	
Base DWF issue heuristic	Majority
PDOM warp scheduling	round-robin

Table 2. Benchmarks Characteristics: BR, TR and CC present branch instruction frequency, the share of taken branches and maximum allowed CTAs per SM, respectively.

Name	Abbr.	Grid Size	CTA Size	#Insn	BR(%)	TR(%)	CC(%)
Back Propag[21]	BKP	2x(1,64,1)	2x(16,16,1)	2.9M	7	74	4
BFS Graph[21]	BFS	16x(8,1,1)	16x(512,1,1)	1.4M	13	51	4
Needle[21]	NW	2x(1,1,1) 2x(2,1,1) … 2x(31,1,1) (32,1,1)	63x(16,1,1)	12.9M	11	72	7
NN_cuda[21]	NN	4x(938,1,1)	4x(16,16,1)	5.9M	8	75	8
Hotspot[21]	HSPT	(43,43,1)	(16,16,1)	76.2M	8	17	2
Dyn_Proc[21]	DYNP	13x(35,1,1)	13x(256,1,1)	64M	7	27	4
Scan[23]	SCN	(64,1,1)	(256,1,1)	3.6M	18	80	4
Coulumb Potential[22]	CP	(8,32,1)	(16,8,1)	113M	6	99	8
Fast Walsh Transform[23]	FWT	6x(32,1,1) 3x(16,1,1) (128,1,1)	7x(256,1,1) 3x(512,1,1)	11.1M	5	49	4,2
Gaussian[22]	GAS	48x(3,3,1)	48x(16,16,1)	8.2M	4	66	4
Matrix Mul[23]	MTM	(5,8,1)	(16,16,1)	2.4M	2	60	4
Srad[21]	SRAD	3x(8,8,1)	3x(16,16,1)	9.1M	6	94	2,3
StoreGPU[22]	STO	(128,1,1)	(256,1,1)	60.5M	1	76	1

placement of the thread in its original warp to avoid extra register back conflict [2,7] during Register Read stage.

DWF's performance is highly sensitive to the warp-issue policy (referred to as *issue heuristic).* An effective issue heuristic should execute all threads of a CTA at the same pace. Greedy issue heuristics can starve low-priority threads. Consequently, when starved threads are finally scheduled for execution, they may not be able to fill the SIMD width resulting in huge performance drop, referred to as *starvation eddies* [6]. In this work we assume the majority issue heuristic [7] as it outperforms other issue heuristics in DWF.

DWF, on the other hand, does not mask any threads and ready threads are always active. Meantime, DWF cannot guarantee execution with maximum SIMD utilization for any basic block as it does not consider reconvergence points.

3. METHODOLOGY

We modified GPGPU-sim (version 2.1.1b) to simulate and evaluate our models. We configured GPGPU-sim to model NVIDIA's Quadro FX5800 with L1 cache for Constant, Texture and Data memories. We assumed 32 PEs per SM (instead of 8 PEs per SM used in conventional GPUs such as FX5800) to model memory access coalescing more effectively. Meantime, we reduced SM clock rate down to 1/4 to reduce the impact of the extra pressure on off-chip DRAM caused by the four times higher number of PEs per SM. Table 1 shows our baseline configuration.

We included benchmarks from Rodinia[21], CUDA SDK 2.3[23] and benchmarks used in[22]. We use benchmarks exhibiting different behaviors: memory-intensiveness, compute-intensiveness, high and low branch divergence occurrence and with both large and small number of CTAs. We exclude some benchmarks due to compilation and runtime problems. Table 2 shows benchmark characteristics for the applications used in this study.

Note that the number of concurrent CTAs per SM is limited by how effectively SM resources are utilized by benchmarks. In Table 2, CC shows the number of CTAs that can be executed concurrently on an SM in each benchmark. For a specific kernel, this number is limited by the size of following resources: thread pool size, shared memory and register file.

In this work we report performance as measure by IPC (Instruction per Clock). GPUs are high-throughput architectures; therefore an alternative way to measure performance is to report Ops/s (Operations per Second). We do not report Ops/s in the interest of space.

4. MACHINE MODELS

According to CUDA Programming Guides [2], three parameters impact performance in GPUs:

- *Workload Parallelism*: In order to achieve high performance, workloads should have enough parallelism to potentially utilize GPU's computational resources. In workloads launching just a single kernel, the kernel should have enough CTAs to utilize all available SMs of the GPU. Workloads launching multiple concurrent kernels can benefit from the "concurrent kernels" feature of modern GPUs [18] and utilize SMs with CTAs of multiple kernels. Meantime, SMs should have enough resources to exploit the benefits of a well-parallelized workload[1].

- *Branch Divergence*: The entire threads forming a warp execute the same instruction until a branch instruction is reached. Upon executing branch instructions, threads taking different paths in a warp force a performance penalty in GPUs using an SIMT architecture. This penalty is the result of replacing execution of a fully occupied warp with two warps (one for the taken path following the branch, one for the not taken path) with less-than-maximum number of threads. Existing GPUs do this by temporarily inactivating threads, which do not belong to the current diverging path, forcing a decrease in the SIMD utilization.

- *Memory Divergence*: When executing memory instructions, threads of a warp can potentially diverge into two groups; threads that find their data in the cache and threads that miss the data. If a thread of a warp misses the data, the entire warp should wait for fetching the data. To reduce the cost [5] associated with the data

[1] We define a well-parallelized workload as a workload that scales to all SMs and assigns close to maximum allowed CTAs to each SM.

fetch, SM proceeds with executing other ready warps effectively hiding the latency associated with the cache miss. In the absence of enough ready warps, SM stalls and has to stay idle waiting for the missing data.

In order to evaluate how each parameter impacts performance in GPUs, we develop a set of machine models. We introduce twelve machine models represented using the X-Y-Z notation. In this representation X indicates if the machine has limited (LR) or unlimited execution resources (UR). Y indicates the control-flow mechanism used by the machine. Here DC, PC and IC represent DWF, PDOM and ideal control-flow, respectively. Finally, Z represents the memory model used. M indicates non-ideal memory systems, whereas IM models an ideal one. Accordingly we study the following twelve machines: **LR-DC-M, LR-PC-M, LR-DC-IM, LR-PC-IM, UR-DC-M, UR-PC-M, LR-IC-M, LR-IC-IM, UR-DC-IM, UR-PC-IM, UR-IC-M** and **UR-IC-IM**.

Machines using **UR** in their notation exploit SMs having unlimited resources to let the GPU interleave execution of all CTAs of the workload concurrently. Under this model there is no resource shortage in "thread pool", "Shared Memory" and "Register File". The inverse notation **LR**, indicates that SMs have the limited resources presented in Table 1. **DC** and **PC**, on the other hand, indicate that a model uses one of the two SIMD control-flow mechanisms (**DC** for **DWF** and **PC** for **PDOM**). Inversely, **IC** indicates using an ideal control-flow mechanism where diverging branches do not impose any additional penalty in the SMs. An IC machine is similar to an MIMD architecture where an SM is a multi-core processor and each PE is capable of executing any instruction. We modeled this machine using the MIMD configuration of GPGPU-sim.

M indicates that a model uses a conventional memory system. The inverse notation, **IM,** indicates access to an ideal memory system with zero latency.

LR-PC-M, for example, represents a real GPU with limited execution resources using the PDOM control-flow mechanism and a realistic memory system. **LR-DC-M** is a machine with limited execution resources using the DWF control-flow mechanism and a realistic memory system. **LR-PC-IM** and **LR-DC-IM** are different from LR-PC-M and LR-DC-M as LR-PC-IM and LR-DC-IM come with an ideal memory system. A machine that has unlimited resources per SM, using a realistic memory system and the PDOM control-flow mechanism is referred to as **UR-PC-M**. Similarly **UR-DC-M** is a machine with unlimited SM resources, DWF control-flow mechanism and realistic memory. **LR-IC-IM** is a machine combining zero-latency memory with the MIMD architecture for SMs but still using limited SM resources. In this machine, individual PEs can execute different instructions. **UR-IC-M** is a machine assuming unlimited SM resource and an MIMD SM architecture. **UR-PC-IM** is a machine model using PDOM as its control flow solution but otherwise ideal. Similarly, **UR-DC-IM** is an otherwise ideal machine using the non-ideal DWF control-flow mechanism. Finally, a 100% ideal machine is represented by **UR-IC-IM**.

A. Performance Potentials

The performance gap between different machine models provides valuable insight towards how each parameter could potentially impact performance. There are three performance gaps per control-flow mechanism:

a) The gap between the performance of LR-XC-IM and LR-XC-M, which shows the speedup that can be reached under an ideal memory system. We refer to this gap as *memory potential*.

b) The gap between the performance of LR-IC-M and LR-XC-M, which shows the speedup achievable when the branch divergence is eliminated completely. We refer to this gap as *control potential*.

c) The gap between the performance of UR-XC-M and LR-XC-M models, which shows potential speedup achievable when SM has unlimited resources to interleave execution of all CTAs. We refer to this gap as *resource potential*.

Figure 2 shows how we use machine models to find and interpret the performance potentials. It also shows average relative performance for each model compared to the baseline. We compare LR-XC-M to LR-XC-IM to find memory potential, LR-XC-M to LR-IC-M to find control potential, and LR-XC-M to UR-XC-M to find

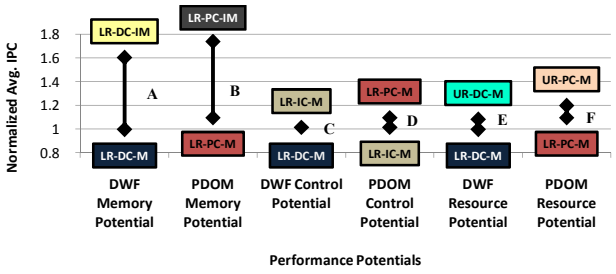

Figure 2. Performance Potentials: *A: DWF Memory Potential:* The gap between performance of LR-DC-M and LR-DC-IM. *B: PDOM Memory Potential:* The gap between performance of LR-PC-M and LR-PC-IM. *C: DWF Control Potential:* The gap between performance of LR-DC-M and LR-IC-M. *D: PDOM Control Potential:* The gap between performance of LR-PC-M and LR-IC-M. *E: DWF Resource Potential:* The gap between performance of LR-DC-M and UR-DC-M. *F: PDOM Resource Potential*: The gap between performance of LR-PC-M and UR-PC-M. The Y-axis is scaled to present relative average performance for all the benchmarks used in this study. Performance is normalized relative to LR-DC-M's performance.

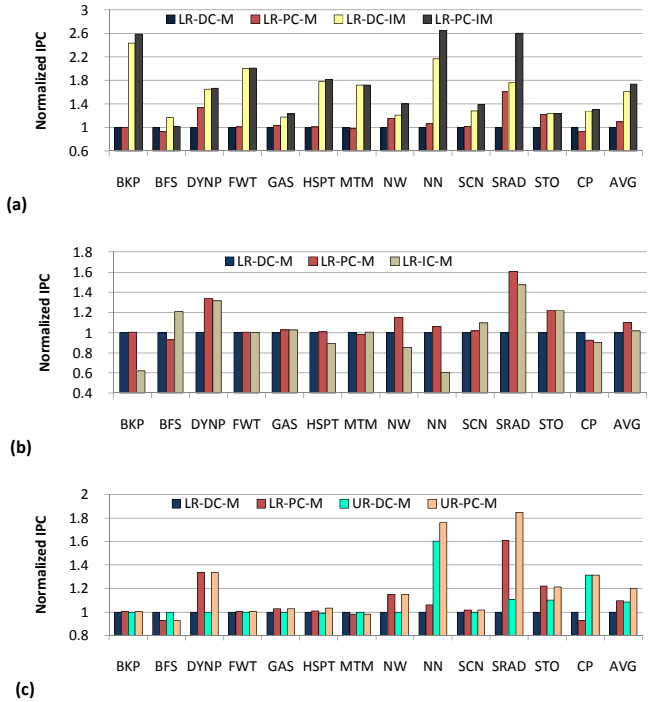

Figure 3. Performance for different machine models showing: (a) Memory potentials. (b) Control potentials. (c) Resource potentials. Performances are presented relative to LR-DC-M.

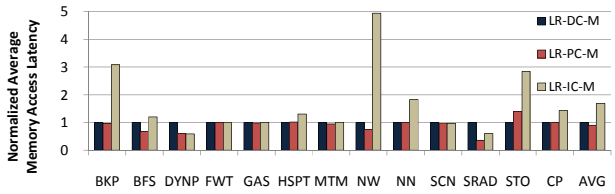

Figure 4. Average memory access latency comparison of 3 machine models involved in control potentials. Average memory access latency is an average latency of all memory accesses of a benchmark. Latencies are normalized to LR-DC-M latency.

resource potential. Memory potential shows potential performance improvement obtainable if the memory system was ideal. Control potential shows potential performance improvement if we could eliminate branch divergence entirely using MIMD architecture instead of SIMD. Resource potential reveals how much performance is enhanced if a GPU could provide enough resources to interleave execution of all CTAs of a well-parallelized workload. These models show the maximum improvement a designer could expect by investing in different GPU components.

Figure 3 shows experimental results for different machine models per benchmark. This figure is organized to show memory potential in 3(a), control potential in 3(b) and resource potential in 3(c). On average, memory potential is largest as removing the memory obstacle improves performance for PDOM and DWF up to 59% and 61% respectively (Figure 3(a)).

As presented in Figure 3(b), eliminating branch divergence results in performance improvement of 2% compared to DWF. Meantime, we witness a performance loss of 7% when comparing a system with ideal control flow to PDOM. Our studies shows that this unexpected behavior could be explained by the fact that exploiting MIMD architecture imposes extra pressure on off-chip DRAMs resulting in an overall performance loss. This pressure is caused by an increase in the number of memory requests resulting from un-coalesced memory accesses occurring under ideal control-flow. To provide better understanding, in Figure 4 we report average memory access latency for the machines presented in Figure 3(b). On average, average memory access latency for MIMD is 87% and 68% higher than SIMD machines using PDOM and DWF, respectively. Similar conclusions could be made regarding DWF's performance. As presented in Figure 3(b), DWF shows lower performance compared to PDOM. This is consistent with Figure 4 where average memory access latency for PDOM is less than DWF. This could be the result of the negative impact of thread regrouping used by DWF on memory access coalescing.

When SMs are equipped with unlimited resources (resource potential), performance is improved by 9.4% and 8.6% for machines using PDOM and DWF respectively (Figure 3(c)). Unlimited resources provide more active threads per SM. Higher number of active threads can facilitate memory access latency hiding. Under a fixed number of CTA limiting resources, DWF already has a higher number of active threads compared to PDOM as it does not mask any threads. Consequently DWF could benefit less from the provided extra resources (thread pool, shared memory and register file) compared to PDOM.

We conclude from Figure 3 that investing in new solutions to address memory inefficiencies would result in much higher returns compared to returns coming from investing in per SM resources or more advanced control-flow mechanisms. Moreover, we conclude that we would benefit from investing in control-flow mechanisms

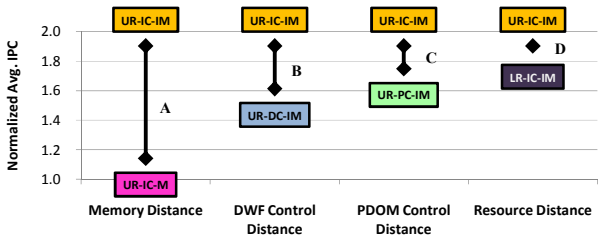

Figure 5. Performance Distances: *A: Memory Distance:* The gap between performance of UR-IC-M and UR-IC-IM. *B: DWF Control Distance:* The gap between performance of UR-DC-IM and UR-IC-IM. *C: PDOM Control Distance:* The gap between performance of UR-PC-IM and UR-IC-IM. *D: Resource Distance:* The gap between performance of LR-IC-IM and UR-IC-IM. The Y-axis is scaled to present relative average performance for all the benchmarks used in this study. Performance is normalized relative to LR-DC-M's performance.

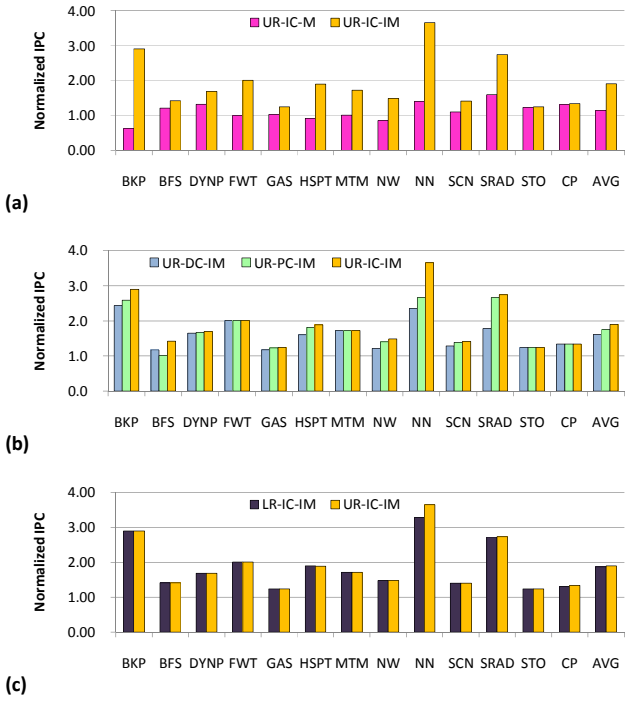

Figure 6. Performance of machine models organized to show (a) Memory distance, (b) Control distance, (c) Resource distance. Performances are reported relative to LR-DC-M.

only if the overall impact on memory pressure is well studied and accommodated.

B. Performance Distances

The models introduced in the previous section help understanding investment points with better and possible returns. We are also interested in finding out which issues keep us further from an ideal GPU. We refer to this as the *performance distance* associated with each parameter. It should be noted that the two answers could be different from a theoretical point of view. The *performance distance of each parameter* can be obtained by finding the gap between the performance of a 100% ideal system (UR-IC-IM) and machines that are under realistic restrictions in only one of the three parameters:

a) The gap between the performance of UR-IC-M and UR-IC-IM shows how much a realistic memory system is responsible for distancing an otherwise ideal system from an ideal performance. We refer to this as *memory distance*.

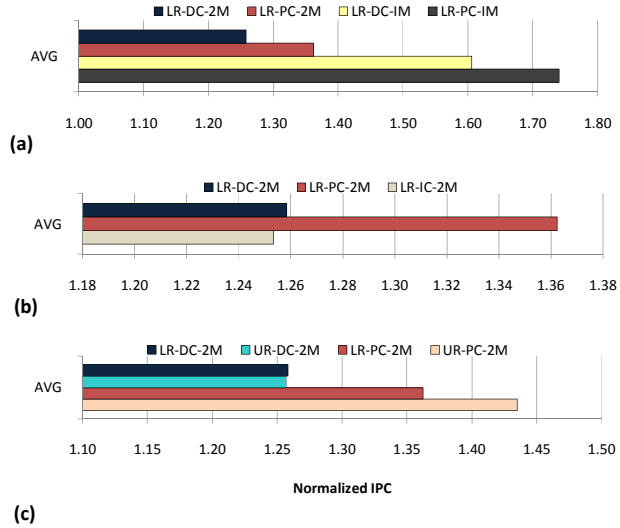

Figure 7. Performance of aggressive memory machine models. (a) Memory potentials. (b) Control potentials. (c) Resource potentials. 2M refers to a machine whose L1 cache size and number of memory controllers are doubled relative to M baseline machine model. Performances are reported relative to LR-DC-M.

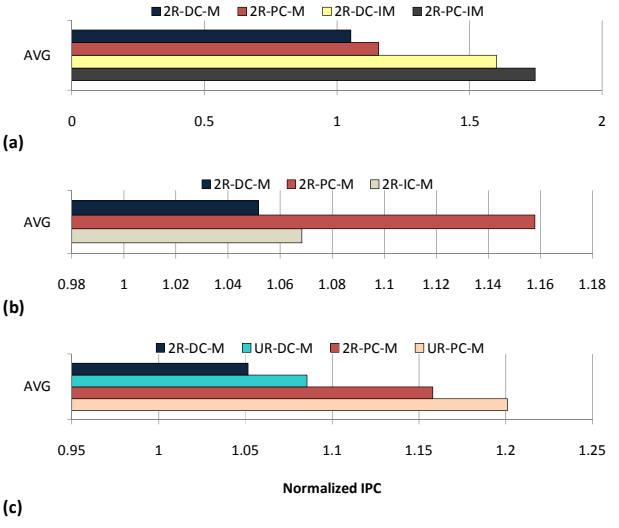

Figure 8. Performance of aggressive resource machine models. (a) Memory potentials. (b) Control potentials. (c) Resource potentials. 2R refers to a machine exploits double thread pool, register file and shared memory relative to LR baseline machine model. Performances are reported relative to LR-DC-M.

b) The gap between the performance of UR-XC-IM and UR-IC-IM shows how much current control-flow mechanisms distance us from an ideal system. We refer to this as *control distance*.

c) The gap between the performance of LR-IC-IM and UR-IC-IM shows how much lack of the resources for interleaving execution of all CTAs downgrades performance compared to an ideal system. We refer to this as *resource distance*.

Figure 5 presents the above distances, and clarifies how much a performance-impacting parameter solely affects GPU performance.

Figure 6 shows experimental results for different machine models per benchmark. This figure is organized to show memory distance in 6(a), control distance in 6(b) and resource distance in 6(c). As reported an ideal system loses 40% performance if forced to exploit a realistic memory system (Figure 6(a)). The control-flow mechanism comes second as an ideal system loses about 15% under DWF and 8% under PDOM (Figure 6(b)). An ideal system shows very little sensitivity to per SM resources as it loses close to 2% performance to resource restrictions (Figure 6(c)). A general conclusion could be made that a non-ideal memory system will impose huge penalties negating most achievements possible by using ideal control-flow or unlimited per SM resources. On the other hand, in the presence of an ideal memory system, the performance lost to realistic control-flow or per SM resources could be compensated to a large extend.

5. SENSITIVITY ANALYSIS

In this section we investigate if our findings presented earlier stay valid under possible variations in GPU microarchitecture. In Subsection A, we study the impact of using a more aggressive memory system on our findings. In Subsection B we study how doubling the number of per SM resources impacts results. In this section we limit our study to performance potentials.

A. Aggressive Memory

We investigate a GPU equipped with an aggressive memory (referred to as LR-XC-2M). This aggressive memory system has twice L1 constant, data and texture cache size and comes with twice the number of memory controllers compared to LR-XC-M. Aggressive memory configuration accelerates LR-DC-M and LR-PC-M up to 26% and 24% respectively (not presented in figures).

In Figure 7 we report average performance potentials as performance gaps in different models. We report memory potential, control potential and resource potential in parts a, b and c respectively.

As presented in Figure 7(a), memory potential is reduced to 28% for both DWF and PDOM under aggressive memory. As Figure 7(b) shows control potential does not improve under aggressive memory; it reaches almost -0.4% for DWF and -8% for PDOM. Resource potential shown in Figure 7(c) is 8% for PDOM and almost zero for DWF. We make the following conclusions from Figure 7: 1) Exploiting an aggressive memory system can reduce memory potential significantly. 2) Having an aggressive memory makes DWF needless to additional resources for interleaving of all CTAs. 3) DWF's performance appears to be more sensitive to memory configuration compared to PDOM.

B. Aggressive Per SM Resources

In this section we investigate our machine models for an SM equipped with aggressive resources. The machines with aggressive resources (which use 2R in their abbreviation) exploits "thread pool", "register file" and "shared memory" resources twice the size of baseline architecture presented in Table 1. Aggressive resources for LR-DC-M and LR-PC-M can accelerate both machines up to 5% (not presented in figures.)

As shown in Figure 8(a), using aggressive resources reduces memory potential down to 52% and 51% for DWF and PDOM respectively. Control potential shows very little sensitivity to aggressive resources; control potential is -8% and 2% for PDOM and DWF, respectively (Figure 8(b)). As reported in Figure 8(c), expectedly, resource potential decreases by equipping baseline architecture with aggressive resources and reaches 4% for PDOM and 3% for DWF.

6. CONCLUDING REMARKS

In this study we developed and studied a set of machine models to identify how much speedup can be expected by eliminating different performance obstacles in GPUs. Moreover, we investigated how much each obstacle contributes to our current distance from an ideal GPU. Our results show that memory has the greatest impact in both measurements.

The GPUs studied use two non-ideal control-flow mechanisms; Dynamic Warp Formulation and Postdomonator Reconvergence. We showed that DWF is more sensitive to branch divergence and memory access latency while it can tolerate resource limitation better than PDOM.

Our findings show that further improvement of control-flow mechanisms requires paying attention to the impact on memory pressure.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] NVIDIA Tesla C2050 Specifications. http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html

[2] NVIDIA Corp. CUDA C Best Practices Guide Version 3.2. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf

[3] J. E. Lindholm et al. NVIDIA Tesla: a Unified Graphics and Computing Architecture. IEEE Micro, Volume 28 Issue 2, March 2008.

[4] Tianhe-1A, Nebulae and TSUBAME 2.0 supercomputers. http://www.top500.org/

[5] H. Wong et al. Demystifying GPU Microarchitecture through Microbenchmarking. IEEE International Symposium on Performance Analysis of Systems and Software, March 2010.

[6] W. Fung and T.M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA-17), 2011.

[7] W. Fung et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture.

[8] J. Meng et al. Dynamic warp subdivision for integrated branch and memory divergence tolerance. Proceedings of the 37th annual international symposium on Computer architecture.

[9] B. W. Coon et al; US Patent 7,353,369: System and Method for Managing Divergent Threads in a SIMD Architecture (Assignee NVIDIA Corp.), April 2008.

[10] Victor W. Lee et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. Proceedings of the 37th annual international symposium on Computer architecture.

[11] B. Coutinho et al. Performance Debugging of GPGPU Applications with the Divergence Map. 2010 International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).

[12] W.M. Hwu et al. Compute Unified Device Architecture Application Suitability. Computing in Science and Engineering, Volume 11, Issue 3, May 2009.

[13] NVIDIA Corp. NVIDIA GeForce 8800 GPU Architecture Overview. pp 38. November 2006.

[14] S. Ryoo et al. Program optimization space pruning for a multithreaded gpu. Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization.

[15] S. Che et al. A performance study of general-purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing, Volume 68 Issue 10, October, 2008.

[16] A. Bakhoda and T.M. Aamodt. Extending the Scalability of Single Chip Stream Processors with On-chip Caches. 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI 2008) in conjunction with ISCA 2008.

[17] D. Tarjan et al. Increasing memory miss tolerance for SIMD cores. Proceeding SC '09, Proceedings of the Conference on High Performance Computing.

[18] NVIDIA Corp. NVIDIA's Next Generation CUDA Compute Architecture: Fermi; v1.1

[19] G. L. Yuan et al. Complexity effective memory access scheduling for many-core accelerator architectures. Proceedings of the 42nd Annual IEEE/ACM International.

[20] N. B. Lakshminarayana and H. Kim.; "Effect of Instruction Fetch and Memory Scheduling on GPU Performance" Workshop on Language, Compiler, and Architecture Support for GPGPU, in conjunction with HPCA/PPoPP 2010, 2010.

[21] Che S, et al.; Rodinia: A benchmark suite for heterogeneous computing. 2009 IEEE International Symposium on Workload Characterization (IISWC). 2009(c):44-54.

[22] Bakhoda A, et al.; Analyzing CUDA workloads using a detailed GPU simulator. 2009 IEEE International Symposium on Performance Analysis of Systems and Software. 2009:163-174.

[23] NVIDIA CUDA SDK. http://developer.nvidia.com/cuda-toolkit-23-downloads