# Microarchitectural Performance Characterization of Irregular GPU Kernels

Molly A. O'Neil
Department of Computer Science
Texas State University
San Marcos, TX
moneil@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, TX
burtscher@txstate.edu

***Abstract*—GPUs are increasingly being used to accelerate general-purpose applications, including applications with data-dependent, irregular memory access patterns and control flow. However, relatively little is known about the behavior of irregular GPU codes, and there has been minimal effort to quantify the ways in which they differ from regular GPGPU applications. We examine the behavior of a suite of optimized irregular CUDA applications on a cycle-accurate GPU simulator. We characterize the performance bottlenecks in each program and connect source code with microarchitectural characteristics. We also assess the impact of improvements in cache and DRAM bandwidth and latency and discuss the implications for GPU architecture design. We find that, while irregular graph codes exhibit significantly more underutilized execution cycles due to branch divergence, load imbalance, and synchronization overhead than regular programs, these factors contribute less to performance degradation than we expected. It appears that code optimizations are often able to effectively address these performance hurdles. Insufficient bandwidth and long memory latency are the biggest limiters of performance. Surprisingly, we find that applications with irregular memory access patterns are more sensitive to changes in L2 latency and bandwidth than DRAM latency and bandwidth.**

## I. Introduction

The last several years have seen graphics processing units (GPUs) appear as general-purpose computation accelerators in many high-performance platforms. For programs that are well suited, GPUs offer a large advantage over multicore CPUs [32] in terms of performance, performance per dollar, performance per transistor, and often energy efficiency [18].

GPUs are very effective at accelerating *regular* programs that access vector- or matrix-based data structures in statically predictable ways. Such applications tend to exhibit large amounts of data parallelism, require little synchronization, and map easily to GPU hardware. GPU implementations of these regular algorithms can be tens of times faster than tuned parallel CPU versions [9].

However, many problem domains employ algorithms that are *irregular* in nature: they build, traverse, and update pointer-based data structures such as trees or graphs and exhibit input-dependent control flow and memory access patterns. Important irregular applications can be found across diverse domains including *n*-body simulation [3], optimization theory [12], meshing [11], satisfiability problems [5], compilers [1], and social networks [17]. Irregular codes do not map to GPU hardware as naturally as regular programs. Nonetheless, the literature includes several GPU implementations of irregular algorithms that outperform their multicore CPU counterparts [6][25][27].

Due to the advantages they offer over multicore CPUs, GPUs are likely to continue to grow in prevalence as accelerators for general-purpose computation. Since there is significant evidence that they are capable of accelerating even irregular applications, it is important to understand the specific demands that irregular codes place on GPU hardware and the ways in which the behaviors of these codes differ from regular programs. This knowledge can guide developers of irregular GPU programs as well as hardware designers hoping to broaden the acceleration capabilities of GPUs.

This paper makes the following main contributions.

- We present the first simulation-based workload characterization focusing on irregular GPU applications.
- We quantify the impact of the control flow and memory access irregularity in these codes on memory coalescing, branch divergence, and cache effectiveness.
- We assess the sensitivity of these applications to cache and DRAM latency and bandwidth, cache size, and coalescing behavior.
- We analyze programs from the irregular LonestarGPU suite [7] and identify relationships between source code and microarchitectural performance behavior.
- We demonstrate that the LonestarGPU codes comprise a wide variety of irregular behavior and performance bottlenecks not typically found in regular GPU code.

The rest of the paper is organized as follows. Section II reviews the concept of irregularity and the challenges it presents to GPU programming. Section III discusses the benchmarks and inputs we study, and it describes the cycle-accurate simulator we employ and its configurations. Section IV discusses and analyzes the results. Section V summarizes prior work. Section VI presents our conclusions.

## II. Background

This section summarizes the differences between regular and irregular code. Regular code refers to programs in which neither control flow nor memory addresses are data dependent. For instance, the dynamic behavior (*i.e.*, the conditional branch decisions and memory reference stream on an in-order processor) of a matrix multiply program can be statically determined based only on the input size and data-structure location but without knowing any input *values*.

Irregular code, in contrast, refers to programs in which the runtime behavior is determined by the input values. Both control flow and memory access patterns may differ for different inputs. Irregular code usually arises from the use of dynamic data structures such as trees and graphs.

Irregular algorithms are more difficult to parallelize in general and more challenging to map to GPUs in particular than regular algorithms. For best performance, GPUs require coalesced memory accesses and identical control flow paths for the threads within a warp. The data-dependent dynamic behavior of irregular codes makes it difficult to assign work to threads in a manner that ensures coalesced accesses, identical control flow, and load balance.

## III. Experimental Methodology

This section describes the applications we study as well as the inputs we choose for each benchmark. Additionally, it details the operation of GPGPU-Sim and the simulator configuration options we apply in our study.

### *A. Applications and Inputs*

We characterize performance aspects of the latest version of the LonestarGPU benchmark suite [23] as of 5/1/2014 in addition to several other programs. LonestarGPU is a collection of hand-optimized CUDA implementations of real-world irregular applications. It includes the following codes.

- Breadth-First Search (BFS): This kernel labels each node in a graph with the node's minimum level (or number of edges) from a specified start node, which is defined to be at level zero [27]. It is a key kernel in many applications such as mesh partitioning.
- Barnes-Hut (BH): This *n*-body algorithm simulates the effect of gravity on a star cluster [3] by, in each time step, hierarchically decomposing the space around the stars, which is recorded in an octree. Special octree traversals allow for the quick approximation of forces.
- Delaunay Mesh Refinement (DMR): This is a mesh refinement algorithm from computational geometry [21] that iteratively transforms the 'bad' triangles of a triangulated input mesh into 'good' triangles by retriangulating the cavity around each bad triangle.
- Minimum Spanning Tree (MST): Boruvka's MST algorithm computes a minimal spanning tree of an input graph through successive application of minimum weight edge contractions. This process is repeated until the graph consists of just a single node.
- Single-Source Shortest Paths (SSSP): This classic graph problem is similar to BFS and computes the shortest path to each node from a designated source node in a directed, weighted graph [12].

LonestarGPU version 2.0 relies on the CUB [13] library of parallel primitives, iterators, and I/O utilities, which in turn is dependent on CUDA 5.5. GPGPU-Sim, the microarchitectural simulator on which we perform our analysis, does not support CUDA versions later than 4.2 as of this writing. We have modified two of the LonestarGPU benchmarks, DMR and MST, to inline the PTX equivalent of their CUB functionality in order to allow them to be run on the simulator. We omit LonestarGPU's Survey Propagation benchmark because it would require more significant modifications to remove its reliance on CUB. We also omit the Points-To Analysis benchmark due to excessive simulation time on even the smallest available input. LonestarGPU includes multiple implementations of several of the algorithms; we characterize the primary implementation in each case.

In addition to the above applications, we also examine the following, more regular GPU programs for comparison.

- Floating-Point Compression (FPC): This program implements a lossless data (de-)compression algorithm for double-precision floating-point values [30]. The code processes chunks of input in parallel. Its control flow depends on how well each word can be compressed.
- Traveling Salesman Problem (TSP): This is a classic optimization problem involving finding the minimal Hamiltonian tour in a complete, undirected graph. The GPU code implements an iterative hill climbing approach with random restarts to find a near-optimal tour [31]. It has a data-dependent memory access pattern.
- N-Body (NB): Similar to BH, this is an *n*-body program that simulates the motion of stars. Unlike BH, it performs precise all-to-all force calculations, making it regular. It is our own implementation and outperforms the corresponding CUDA SDK code.
- Monte Carlo (MC): This program evaluates the fair call price for a set of European options using the Monte Carlo method. It is a highly regular, array-based code from the CUDA SDK version 4.2 [33].

Table 1 lists the inputs we use. For each application, we select an input large enough to keep the simulated hardware busy but small enough to result in reasonable (less than 48-hour where possible) simulator runtimes. For each irregular code, Table 1 also includes the size of the data structures read by the dominant kernel's inner loop (the working set) compared to the simulated GPU's L2 cache size. By definition, irregular codes display input-dependent behavior. We have attempted to select a realistic, representative input for each program. Additionally, we study a second input for each application to quantify the expected behavior variation. Input variance is discussed in Section IV.C. Table 1 lists the primary input first.

Table 1: Application inputs

| Code | Input |
|---|---|
| BFS | NYC road network (~264K nodes, ~734K edges) |
| | *(working set = 3898 kB = 5.08x L2 size)* |
| | RMAT graph (250K nodes, 500K edges) |
| BH | 494K bodies, 1 time step |
| | *(working set = 7718 kB = 10.05x L2 size)* |
| DMR | 50.4K nodes, ~100.3K triangles, maxfactor = 10 |
| | *(working set w/ maxfactor 10 = 7840 kB = 10.2x L2 size)* |
| | 30K nodes, 60K triangles |
| MST | NYC road network (~264K nodes, ~734K edges) |
| | *(working set = 3898 kB = 5.08x L2 size)* |
| | RMAT graph (250K nodes, 500K edges) |
| SSSP | NYC road network (~264K nodes, ~734K edges) |
| | *(working set = 3898 kB = 5.08x L2 size)* |
| | RMAT graph (250K nodes, 500K edges) |
| FPC | obs_error dataset (60 MB), 30 blocks, 24 warps/block |
| | num_plasma dataset (34 MB), 30 blocks, 24 warps/block |
| TSP | att48 (48 cities, 15K climbers) |
| | eil51 (51 cities, 15K climbers) |
| NB | 23,040 bodies, 1 time step |
| MC | 256 options |

### *B. GPGPU-Sim*

To characterize the performance of the selected irregular GPU codes, we study the benchmarks using GPGPU-Sim, a cycle-accurate microarchitectural model of an NVIDIA-like GPU for general-purpose computation [2]. GPGPU-Sim models the streaming multiprocessors (SMs), L1 caches, texture and constant caches, shared memory, interconnect network, memory partition (including the L2 cache), and off-chip DRAM. In addition to the caches and shared memory, each SM models an instruction cache, fetch, decode, an issue scheduler, the SIMT stacks used to resolve branch divergence, a scoreboard and operand collector for register file access, as well as ALU and load/store pipelines. GPGPU-Sim models shared memory bank conflicts and coalescing stalls based on the coalescing logic of compute capability 1.3 devices.

GPGPU-Sim performs functional and timing simulation of PTX assembly instructions extracted from a CUDA executable. In PTXPlus mode, it has been correlated against an NVIDIA GT 200 GPU with an IPC correlation of 97.6% on a subset of the CUDA SDK and against Fermi (compute capability 2.0) hardware with an IPC correlation of 97.3% on the Rodinia benchmark suite with reduced problem sizes [16]. Because PTX is a virtual ISA and not the actual code that runs on the hardware, GPGPU-Sim supports PTXPlus, an extended version of PTX that adds addressing modes, condition codes, and instructions similar to those in SASS, NVIDIA's native hardware assembly. While PTXPlus simulation is likely to result in more accurate correlation to real hardware, GPGPU-Sim does not yet fully support it for all programs supported in PTX. Many of our benchmarks do not run successfully in PTXPlus mode. PTX assumes an infinite register set and thus simulation results do not capture the impact of register spill code. Of our studied codes, however, only NB causes register spills in its dominant kernel.

All of the studied codes were compiled using CUDA 4.2. We performed our experiments using GPGPU-Sim version 3.2.1 with a handful of minor bug fixes and instrumented with additional performance counters.

### *C. Simulator Configurations*

GPGPU-Sim release 3.2.1 includes a configuration for the GTX 480, a Fermi NVIDIA GPU [14]. The GTX 480 has 15 SMs. Each SM includes two warp schedulers and two dispatch units, allowing warps to dual-issue. The 32 threads in a warp are issued over two cycles, 16 threads at a time. We use the GTX 480 configuration as the default configuration in our study. Several of the studied benchmarks utilize the CUDA API routine cudaFuncSetCacheConfig, which sets (on a kernel basis) the L1 cache configuration to either 48kB software-controlled shared memory and 16kB hardware-managed cache or vice-versa. GPGPU-Sim v3.2.1 does not support this API call. Hence, we modified the benchmark codes to remove these calls and ran each benchmark with the shared memory and L1 data cache size set according to the program's dominant kernels. We confirmed on real hardware that these changes had a negligible impact on runtime.

Table 2: Simulator configurations

| | Latency | | Bus width | | L1D | | | | | | L2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ROP | DRAM | Ict | DRAM | CP | Sz (PS) | Sz (PL) | MQ | MS | MM | Size | MQ | MS | MM |
| Default | 240 | 200 | 32 | 4 | Y | 16 | 48 | 8 | 32 | 8 | 768 | 4 | 32 | 4 |
| 1/2x ROP | 120 | 200 | " | " | " | " | " | " | " | " | " | " | " | " |
| 2x ROP | 480 | 200 | " | " | " | " | " | " | " | " | " | " | " | " |
| 1/2x DRAM | 240 | 100 | " | " | " | " | " | " | " | " | " | " | " | " |
| 2x DRAM | 240 | 400 | " | " | " | " | " | " | " | " | " | " | " | " |
| No Latency | 0 | 0 | " | " | " | " | " | " | " | " | " | " | " | " |
| 1/2x L1D Cache | 240 | 200 | 32 | 4 | Y | 8 | 24 | 8 | 32 | 8 | 768 | 4 | 32 | 4 |
| 2x L1D Cache | " | " | " | " | " | 32 | 96 | " | " | " | " | " | " | " |
| 1/2x L2 Cache | " | " | " | " | " | 16 | 48 | " | " | " | 384 | " | " | " |
| 2x L2 Cache | " | " | " | " | " | " | " | " | " | " | 1536 | " | " | " |
| 1/2x DRAM Bandwidth | 240 | 200 | " | 2 | Y | 16 | 48 | 8 | 32 | 8 | 768 | 4 | 32 | 4 |
| 2x DRAM Bandwidth | " | " | " | 8 | " | " | " | " | " | " | " | " | " | " |
| 1/2x Ict + DRAM B/W | " | " | 16 | 2 | " | " | " | " | " | " | " | " | " | " |
| 2x Ict + DRAM B/W | " | " | 64 | 8 | " | " | " | " | " | " | " | " | " | " |
| No Coalesce Penalty | 240 | 200 | 32 | 4 | N | 16 | 48 | 8 | 32 | 8 | 768 | 4 | 32 | 4 |
| NCP + Impr L1 Miss | " | " | " | " | N | " | " | 16 | 64 | 16 | " | 4 | 32 | 4 |
| NCP +Impr L1+L2 Miss | " | " | " | " | N | " | " | 16 | 64 | 16 | " | 8 | 64 | 8 |

Latencies represent number of shader core cycles. Cache sizes in kB. ROP=Raster Operations Pipeline (models L2 hit latency). Ict = Interconnect (flit size). CP=Coalesce penalty, PS = Prefer Shared Mem, PL = Prefer L1, MQ=Miss queue entries, MS=Miss status holding register entries, MM=Max MSHR merges

In addition to the default configuration, we modified the provided GTX 480 configuration to scale the minimum L2 hit latency, minimum DRAM latency, interconnect bandwidth, DRAM bus width, the L1D and L2 cache sizes, and the number of cache miss queue and miss-status handling register (MSHR) entries. Moreover, we added a configuration option to GPGPU-Sim to disable the pipeline penalty associated with uncoalesced memory accesses within a warp. Table 2 summarizes the simulator configurations used.

## IV. Results and Analysis

This section first studies the impact on the selected codes of several common sources of GPU performance limitation. Then we examine each application individually and assess the dominant performance bottlenecks of each.

The LonestarGPU benchmarks include applications across a wide range of performance points. Figure 1 illustrates the instructions per cycle (IPC) of each studied application. The theoretical peak performance of the GTX 480 is 480 IPC.

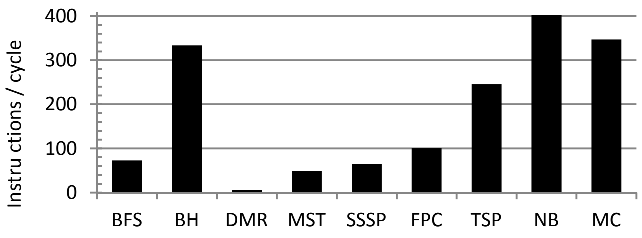


Figure 1: Measured instructions per cycle of each benchmark

As expected, the regular programs NB and MC perform very well. More surprisingly, the irregular code BH also reaches a high IPC, though not in all of its kernels (see below). The remaining irregular codes underperform even FPC, the worst of the (semi-)regular codes. DMR's IPC is dismal for reasons we explain in the individual application analysis below. Overall, there is a clear tendency towards much lower IPCs for irregular codes and none of them come close to achieving peak performance. However, it is also clear that there is no simple or fixed delineation between the performance of codes operating on irregular data structures and more regular codes.

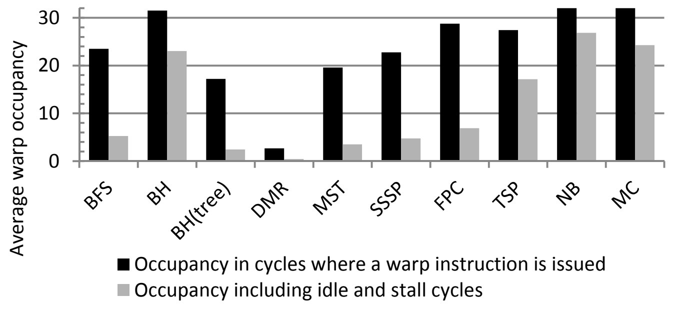
### A. Common Performance Bottlenecks

GPU performance is heavily impacted by the presence of control flow divergence within a warp and memory accesses that cannot be coalesced. We study the effect of both of these factors on each code. In addition, we examine the cache and memory performance of each application, including the effect of cache size as well as cache hit and main memory latency and bandwidth.

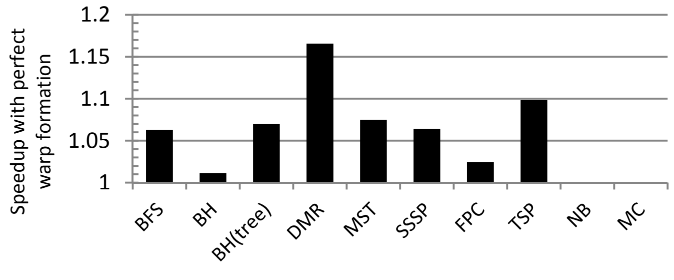
#### 1) Control flow divergence

Figure 2(a) plots the average warp occupancy based on the active mask of instructions within a warp at the issue stage in each scheduler. The first bar for each benchmark represents the average warp occupancy in only those cycles where a warp instruction was issued to the core pipeline. The second bar represents the average warp occupancy across all cycles of the simulation, including cycles in which a scheduler did not issue any warp due to a stall or idle condition.

The left bar for each application provides a graphical representation of the amount of branch divergence. A program with no control flow divergence would reach an average warp occupancy (not including idle and stall cycles) of 32. Indeed, the two highly regular codes we study have average warp occupancies very close to 32, implying they do not suffer from branch divergence. As we expect from codes operating on an irregular data structure, BFS, MST, SSSP, and especially DMR display lower warp occupancies, but only DMR's occupancy falls below half-occupied. Of the irregular codes, only BH has very little control flow divergence because the force calculation kernel, which accounts for 95% of the runtime, has been implemented in a warp-based manner to improve performance. In contrast, the tree-building kernel, denoted as 'BH(tree)', exhibits significantly more control irregularity. Overall, with the exception of DMR, the branch divergence is not as severe as we had expected from irregular codes. Figure 2(b) illustrates the speedup that each benchmark would achieve given perfect warp formation, *i.e.*, if each cycle in which an issue could be made issued 32 instructions per scheduler.



(a)



(b)

Figure 2: (a) Average warp occupancy of each application, both inclusive and exclusive of idle/stall cycles; (b) Benchmark speedup with perfect warp formation

The right bar of each application in Figure 2(a) illustrates the impact of issue stalls on warp occupancy, particularly due to memory latency and uncoalesced accesses. The rest of this subsection investigates these factors in more detail.

#### 2) Memory coalescing

Figure 3(a) plots the average number of memory accesses performed by each global or local load or store instruction. A bar height above one illustrates the presence of uncoalesced memory accesses. The highly regular applications NB and MC have average access counts of 1, meaning that essentially all loads and stores are fully coalesced, *i.e.*, each access by a warp results in a single memory transaction. BH and TSP perform very few but quite uncoalesced stores. On average, BH's loads are almost all coalesced, but this is again due to the dominant 'regularized' force calculation kernel. The BH tree building kernel, in contrast, exhibits more uncoalesced loads than any other application. TSP possesses a data-dependent and byte-granular memory access pattern and thus exhibits highly uncoalesced accesses. FPC, though a semi-regular code, also suffers from a high average access count resulting from its data-dependent byte-granular memory accesses, since stores to two bytes in the same word are serialized by the hardware. BFS and SSSP have fully coalesced stores and display only slightly increased load access counts; however, their high load instruction counts result in significant slowdown from coalescing.

To further study the relationship between coalescing and performance in these benchmarks, we added a configuration option to GPGPU-Sim that removes the pipeline stall penalty associated with non-coalesced accesses. This configuration allows an SM to issue a warp instruction requiring multiple

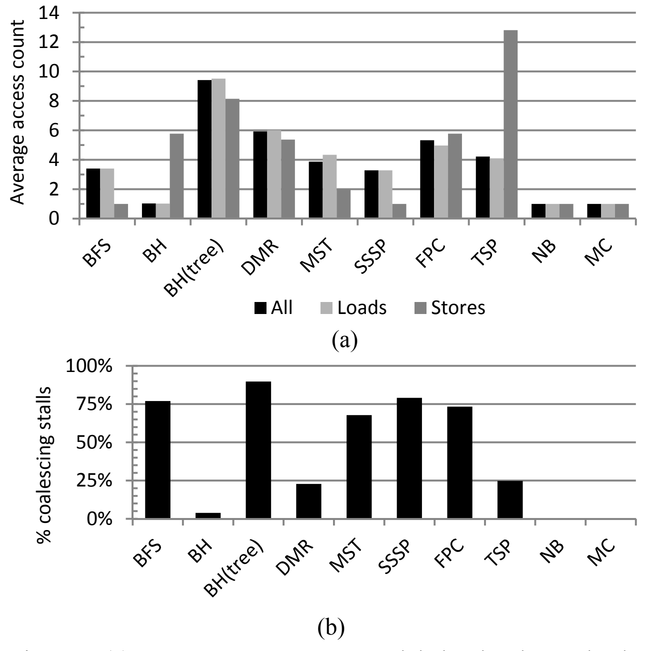


Figure 3: (a) Average access count per global or local warp load or store instruction; (b) Percentage of cycles marked as coalescing stalls

accesses in a single cycle. However, it does not further improve the memory pipeline to handle the increased memory traffic. It is not intended to model a realistic hardware improvement, but it provides some visualization of theoretical improvement. We studied each benchmark with the no-coalesce-penalty configuration applied both by itself and in combination with increased-capacity cache miss queues and MSHRs. Figure 3(b) displays the percentage of simulation cycles in each benchmark that the simulator marks as stalls due to coalescing; this provides a visualization of the theoretical speedup that would result from somehow entirely removing the coalescing requirements. Figure 4 plots the speedups over the default setting for each of our no-coalesce-penalty configurations.

As intuitively expected, simply removing the pipeline penalty associated with coalescing stalls has little impact on performance due to a corresponding increase in cache reservation stalls and interconnect backup. More surprisingly, for most of the benchmarks, improving the miss-handling capability of the caches does little to improve the performance of removing the coalescing penalty. (FPC is an outlier due to its many serialized byte-granular stores, which are counted as coalescing stalls). This suggests that hardware improvements aimed at reducing the coalescing penalty are likely to be ineffective on irregular codes unless they are combined with increased memory bandwidth.

### *3) L2 and DRAM latency*

Next we examine the performance impact of scaling the L2 hit latency and DRAM access latency. GPGPU-Sim models the minimum L2 hit latency via the raster operations pipeline (ROP) latency, which determines the minimum latency between when a memory request arrives at the memory partition and when it accesses the L2 cache. Additionally, the simulator allows configuration of the DRAM latency, the minimum latency between when a memory request accesses the L2 cache and when it is pushed to the DRAM scheduler. The default options for these latencies are based on a microbenchmarking study of the GT200 [34]. We do not assess the effect of scaling the L1 hit latency. GPGPU-Sim models the minimum L1 hit latency as a single cycle. However, a recent study [24] suggests a significantly longer hit latency, making this an area for further investigation.

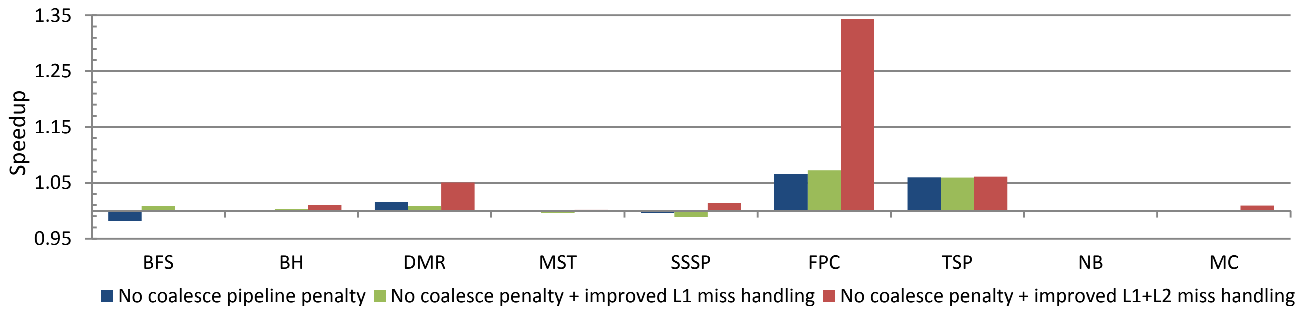


Figure 4: Speedups over the default simulator configuration when removing the coalescing pipeline penalty and increasing the size of the cache miss queues and miss status handling registers

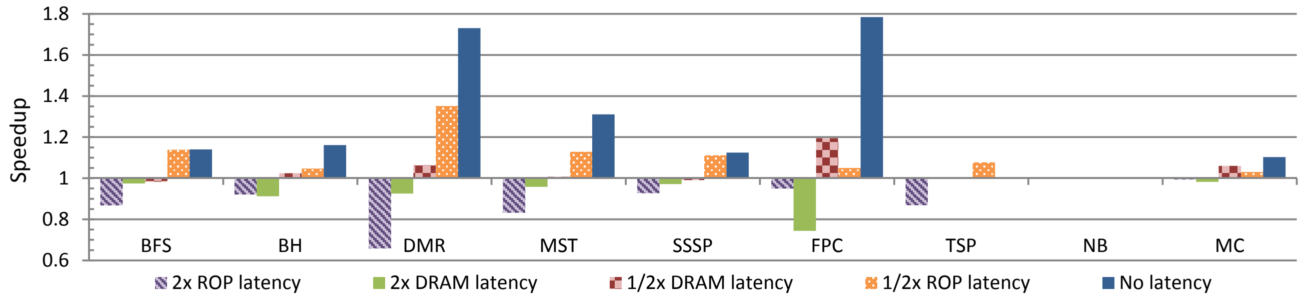


Figure 5: Speedup over the default simulator configuration when scaling the minimum L2 hit latency and DRAM latency
(TSP does not include data for the No Latency configuration due to a simulator deadlock)

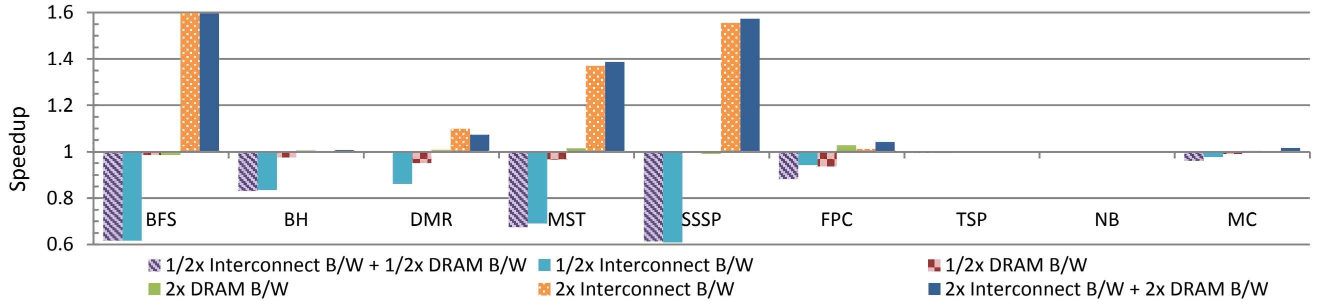


Figure 6: Speedup over the default simulator configuration when scaling the interconnect and DRAM bandwidth
(DMR does not include data for the 1/2X Ict B/W + 1/2X DRAM B/W configuration due to a simulator deadlock)

We examine each benchmark with both latencies configured to zero as well as with the latencies doubled and halved. Figure 5 plots the speedup relative to the default setting for each latency configuration. Interestingly, nearly all of the studied benchmarks are more sensitive to the L2 latency than the DRAM latency, even in the presence of working set sizes several times larger than the L2 capacity. The exception is FPC, which accesses data in a streaming manner and displays high spatial locality. The regular NB code uses tiling to read all data into shared memory and is largely compute-bound; it is thus insensitive to memory latencies. These results suggest that, at least for our inputs, L2 latency is more important than DRAM latency for the performance of GPGPU codes, especially irregular ones.

*4) Interconnect and DRAM bandwidth*

Next we scale the interconnect bandwidth between the memory partitions (including the L2 cache) and the core, as well as the DRAM bandwidth (by adjusting the DRAM bus width). Figure 6 illustrates the performance impact on each benchmark of halving and doubling the interconnect and DRAM bandwidths. Similarly to the L2 and DRAM latency behavior, most of the studied applications are significantly more sensitive to interconnect bandwidth than to DRAM bandwidth. It seems that for these applications and input sizes, the L2 is large enough that sufficient L2 bandwidth keeps enough warps able to execute. The regular codes are helped very little by additional memory bandwidth.

*5) Cache behavior*

Lastly, we observe cache misses per thousand warp instructions (MPKI) for each program, plotted in Figure 7. Most of the applications, including the highly regular codes, have L1 cache miss ratios above 50%, which would be considered extremely high for CPU applications. After all, CPU and GPU architectures have L1 data caches for different reasons: in GPUs, they mostly provide coalescing support rather than exploit temporal locality, because there cannot be an expectation of the cache holding data for a significant period of time due to the high number of active threads.

However, the irregular codes all have markedly higher MPKI rates than the regular codes. NB, which tiles its data into shared memory and is compute-bound, has MPKI rates near zero for both caches. TSP is composed mostly of shared memory accesses and many of its local memory accesses are strided, which is why this program has one of the lowest observed L1 miss rates. FPC also exhibits a relatively low cache miss rate due to its streaming behavior. BFS and SSSP both perform a significant number of pointer-chasing operations and are not able to exploit much spatial locality, resulting in extremely high L1 miss rates.

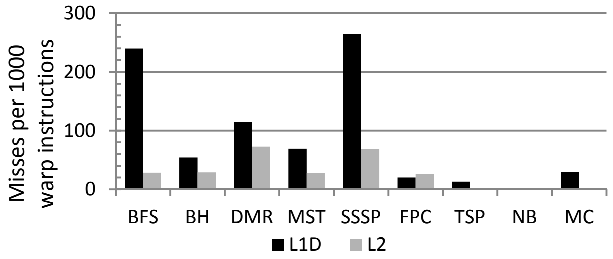


Figure 7: Cache misses per thousand warp instructions (MPKI)

Figure 8 plots the performance impact on each benchmark of halving and doubling the L1D and L2 cache sizes over the default configuration. In general, those benchmarks that are significantly sensitive to interconnect bandwidth also benefit most from increased L1 data cache size. The exception is the irregular BH tree construction kernel. This kernel traverses tree prefixes beginning with the root of the tree. The top of the tree

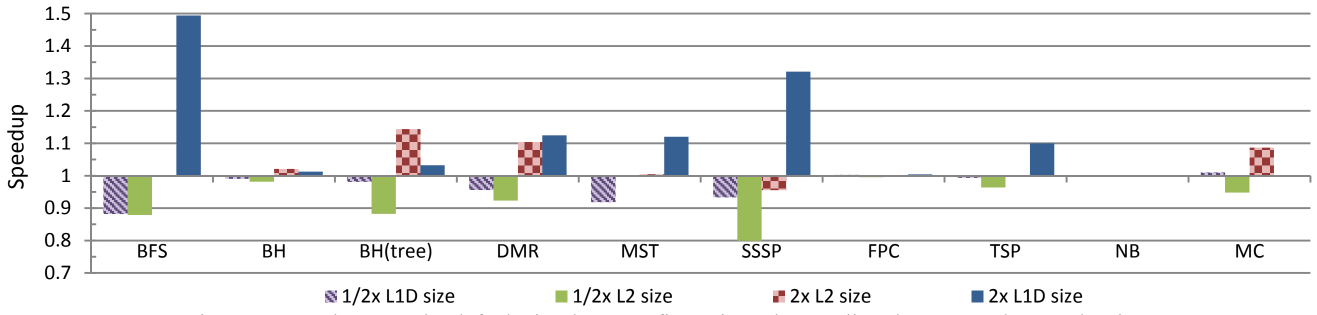


Figure 8: Speedup over the default simulator configuration when scaling the L1D and L2 cache size

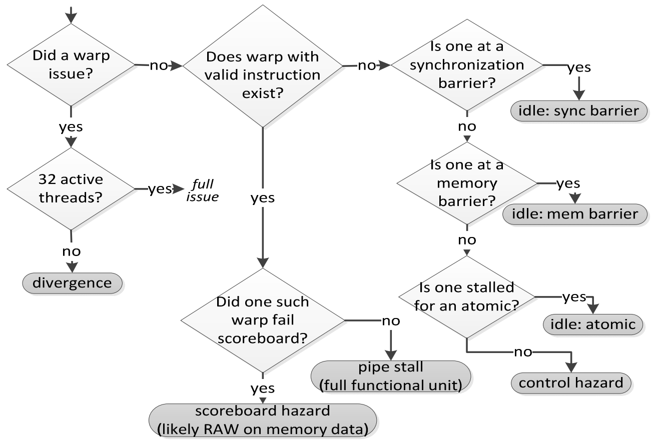
is therefore likely to hit in the L1, but after the top portion of the tree there is insufficient locality to leverage even a larger L1. Increased bandwidth to the L2, however, improves performance because the L2 allows for significant exploitation of locality in the traversals. Nearly all of the codes are hurt significantly more by decreased L2 capacity than decreased L1 capacity. For this input, SSSP displays an unexpected slowdown with an increased L2 size. The SSSP code iterates until convergence is reached, which can cause architectural changes to alter the timing between threads and result in a different number of instructions executed, leading to occasionally counterintuitive performance changes.

## B. Individual Application Analysis

We supplemented GPGPU-Sim's warp issue metrics with additional stall type counters. In each cycle, every SM increments a histogram bin with the active instruction count of the warp it issued in that cycle. In cycles with no issue, a histogram bin for the cause of the issue stall is incremented. To ensure that only one bin is updated per scheduler per cycle, it is necessary to define a priority between idle/stall conditions since warps ineligible for issue in that cycle may be stalled for multiple reasons. Figure 9 illustrates the priority definitions of the issue histogram bins. In case of issue stalls due to a full functional unit pipe, we added instrumentation to collect additional information on the functional unit responsible for the stall. In our codes, the majority of these stalls are due to pressure in the load/store functional unit.

Based on the resulting warp occupancy histogram and stall distribution, we calculated the number of issue cycles with underused thread occupancy due to branch divergence, control flow, barriers, atomics, functional unit stalls, and work imbalance between blocks (denoted as 'interblock imbalance'). Figure 10 displays the breakdown of underused cycles on a per-application basis as well as the cycles in which the GPU issued at full occupancy (which are denoted as 'busy' cycles). Scoreboard collision stalls include both read-after-write (RAW) and write-after-write (WAW) hazards but are dominated by RAW hazards in all of the codes. Because the majority of these RAW hazards are caused by outstanding loads, the 'scoreboard hazards' metric provides a rough estimate of the impact of memory latency. In addition, load/store unit (LSU) pipeline stalls reflect both coalescing penalty and cache reservation fails; the latter is also an indication of memory latency associated slowdown.

We discuss each code in detail and conclude with general observations about the major performance limitations:


Did a warp issue?
no
Does warp with valid instruction exist?
no
Is one at a synchronization barrier?
yes
idle: sync barrier
yes
32 active threads?
yes
full issue
no
divergence
yes
no
Is one at a memory barrier?
yes
idle: mem barrier
no
Is one stalled for an atomic?
yes
idle: atomic
no
control hazard
Did one such warp fail scoreboard?
no
pipe stall (full functional unit)
yes
scoreboard hazard (likely RAW on memory data)
Figure 9: Issue stage histogram bin definitions

- *Breadth-First Search (BFS)*: This code suffers from a high number of LSU and RAW stalls due to the data-dependent nature of its memory accesses (based on the connectivity of the input graph). The BFS implementation further displays some control-flow irregularity due to graph nodes having different numbers of edges, which results in branch divergence.
- *Barnes-Hut (BH)*: The BH code, while tree-based and irregular in nature, is dominated by the force calculation kernel, which has been optimized to eliminate almost all divergence and to ensure that most of its main memory accesses are coalesced. As a result, BH spends a larger percentage of execution time at full occupancy than the other irregular codes. It should be noted that BH's warp threads perform some unnecessary computation to minimize branch divergence, and these unnecessary cycles are denoted as busy cycles rather than divergence. However, the unnecessary work improves both the performance and the accuracy of the algorithm. Figure 10 also includes metrics for the BH tree-building kernel. This kernel exhibits significantly more irregularity, suffering from both memory access stalls and substantial branch divergence. It also exhibits a noticeable performance penalty due to the synchronization barriers necessary to build the tree in parallel. There is a small amount of work imbalance as it is a priori unknown how deep the various branches of the unbalanced octree will be.
- *Delaunay Mesh Refinement (DMR)*: DMR has by far the lowest IPC of all the codes we examine and is in a sense the most irregular code as well. It suffers from a large amount of memory access stalls, divergence, and by far

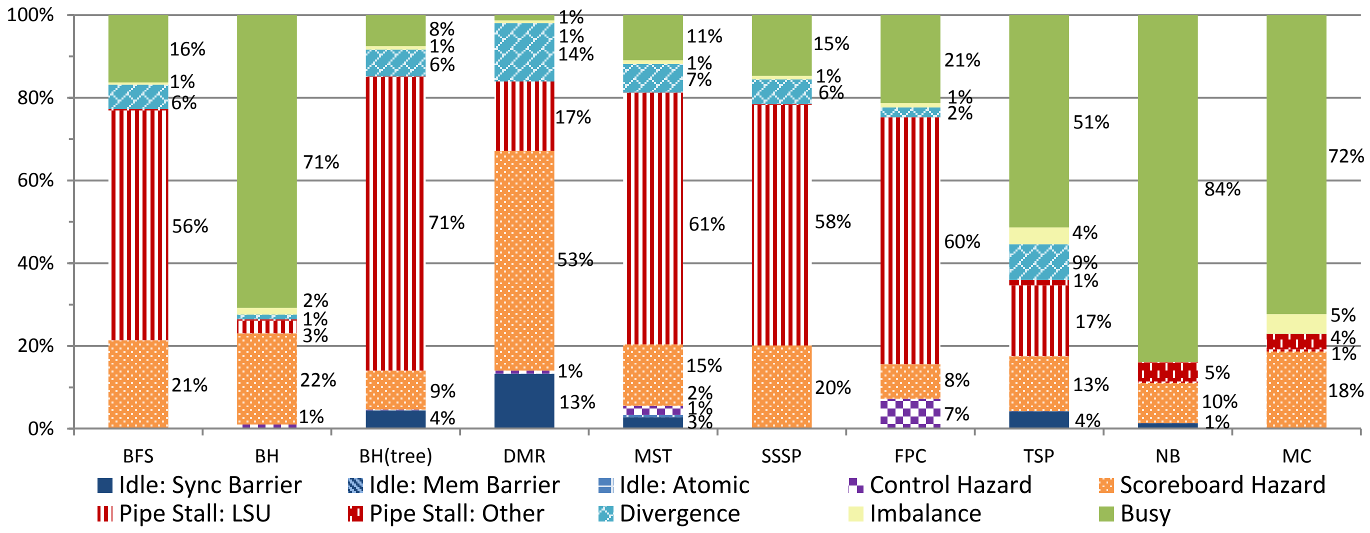


Figure 10: The proportion of underused vs. fully-occupied cycles in each application

the largest fraction of synchronization stalls of any of our codes. For each bad triangle, DMR's refinement kernel builds a cavity whose size and shape are data dependent, checks whether there is overlap with another cavity, employs a priority-based back-off mechanism in case of overlap, and finally refines the cavity if the thread has the highest priority of all the triangles in the cavity. The synchronization stalls stem from global barriers separating these phases, the divergence is the result of load imbalance between threads, and the memory stalls are likely unavoidable when processing an irregular graph whose shape changes at runtime.

- *Minimum Spanning Tree (MST)*: MST spends the majority of its cycles waiting for uncoalesced load data due to the irregular nature of its accesses to the merged graph nodes, or components. The innermost loop contains a set of nested *if* statements to unify the minimum-weight components, resulting in significant divergence penalty and the highest control flow penalty of our irregular codes. MST displays more slowdown associated with atomics than any of our other codes due to the atomic operations necessary to merge components; however, it is still a minor source of performance loss compared to memory-related stalls.
- *Single-Source Shortest Paths (SSSP)*: This algorithm is similar to the BFS implementation. Its performance is also limited by LSU and RAW stalls resulting from uncoalesced memory accesses and insufficient memory bandwidth, as well as branch divergence due to control flow irregularity when processing the input graph.
- *Floating-Point Compression (FPC)*: This application spends most of its time stalled due to a full load/store unit pipeline. These stalls stem from the coalescing behavior of its double-precision memory accesses as well as warp-threads reading and writing data-dependent byte locations in main memory. FPC also exhibits a large percentage of control hazard stalls due to the short (maximum iteration count of 8) data-dependent loops that read and write the compressed bytes corresponding to an uncompressed double. FPC further suffers from some branch divergence due to imbalance in the number of bytes processed by each warp thread, as well as a code section in which only every other thread has work.
- *Traveling Salesman Problem (TSP)*: The TSP code is semi-regular, possessing relatively regular control flow but data-dependent memory accesses, and it spends the majority of its execution time on computation. Its memory accesses are mostly to shared memory, but it does access main memory when performing the 2-opt city ordering swaps, resulting in uncoalesced accesses and LSU stalls. TSP also exhibits a large fraction of idle time associated with synchronization barriers. These barrier idle cycles occur as threads in a block finish computing their locally best solution but have to wait for the slowest thread in the block before the global solution can be updated. The branch divergence stems from an instance of control-flow irregularity where some threads have reached a local minimum and want to move on to a new tour while other threads are still searching for a minimum.
- *N-Body (NB)*: This code is highly regular and computation-bound. Additionally, we chose an input size to exactly fill the resident blocks to provide a basis of comparison for the irregular benchmarks. NB demonstrates the highest busy ratio of the studied programs. Its lost cycles are due mostly to RAW hazards on the small amount of data not accessed via shared memory as well as full computation pipelines and the synchronization barriers necessary between transferring data to shared memory and the computation phase.
- *Monte Carlo (MC)*: The MC application is embarrassingly parallel, highly regular, and dominated by computation. Its largest source of slowdown is scoreboard hazards due to cache misses. It also displays some imbalance between blocks as well as pipeline stalls in the special function unit (SFU) pipeline due to its extensive use of square root operations in the quasi-random sequence generation kernel.

Many of the irregular codes rely on synchronization barriers and atomic operations. Interestingly, these primitives contribute a smaller fraction of program slowdown than expected. The performance loss associated with imbalance and branch divergence was also somewhat less severe than we expected of codes operating on irregular data structures. Clearly, the benchmark codes have been successfully optimized to minimize the performance impact of these aspects [28][29]. In line with expectation, memory bandwidth appears to be the most significant performance bottleneck for highly optimized irregular GPU applications. Apparently, this bottleneck is more difficult to address with source-code optimizations, making corresponding hardware enhancements very valuable for accelerating irregular codes.

### *C. Input sensitivity*

In addition to the primary input listed in Table 1, we compared the behavior of each code for a second input of similar size to understand the impact of input variation on our results. Figure 11 compares the application behavior for two of the graph codes operating on their primary input (a New York City roadmap) and a randomly-generated RMAT graph. The benchmarks BFS and SSSP displayed the highest amount of input variation. For BFS and SSSP, the RMAT input results in significantly more cycles lost to divergence. RMAT graphs are denser and have higher and more varied out-degree per node than road networks, so we would expect the graph traversal codes to exhibit a greater performance penalty associated with many uncoalesced accesses.

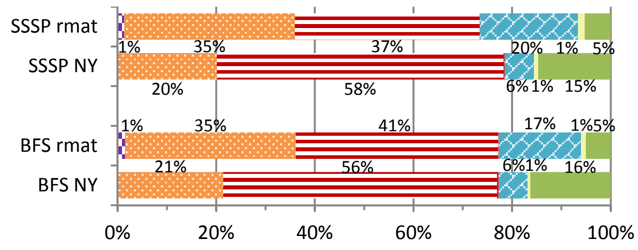


Figure 11: Input variance for BFS and SSSP

## V. Related Work

There have been several prior studies characterizing GPU applications using simulation, but they focus on mostly regular codes. Bakhoda *et al.* [2] present GPGPU-Sim and study twelve CUDA applications (including BFS) demonstrating various levels of GPU performance. They characterize the performance impact of several microarchitectural design choices, including interconnect topology, caches, memory controller design, and workload distribution. Goswami *et al.* [15] propose a set of microarchitecture-agnostic GPGPU workload characterization metrics and use these metrics to study benchmarks in the CUDA SDK, Rodinia, and Parboil suites using GPGPU-Sim. Blem *et al.* [4] propose a set of challenge benchmarks (selected from the GPGPU-Sim benchmarks, Rodinia, and a handful of naïve ports of Parsec applications) where the achieved IPC is less than 40% of peak. They use GPGPU-Sim to present a characterization of the benchmarks' key architectural bottlenecks and apply an analytic model to predict the performance impact of mitigating each bottleneck. Che *et al.* [10] evaluate the Rodinia benchmark suite on a GTX 480 and in GPGPU-Sim.

GPGPU application performance has also been studied using PTX emulators that do not provide cycle-accurate simulation. Kerr *et al.* [19] propose a set of metrics for GPU workloads and analyze these metrics on over fifty mostly regular applications, including the SDK and Parboil, via the GPU Ocelot emulator. They investigate the impact of optimizations such as various branch re-convergence mechanisms and memory read coalescing. Wu *et al.* [35] study several benchmarks (including the SDK, Rodinia, and Parboil) to identify sources of control-flow irregularity.

Burtscher *et al.* [7] previously characterized the control flow and memory access irregularity of the LonestarGPU suite. Their study relied on hardware performance counters for issued and executed instructions, divergent branches, and instructions replayed for coalescing or bank conflicts. Che *et al.* [8] also describe a hardware performance counter-based characterization of a suite of irregular GPGPU graph applications. Many of the counters of interest in a study of irregular codes (*e.g.*, stall cause per issue cycle) are not available through hardware counters, which is why we believe simulation to be necessary to provide a more complete picture of application behavior.

Meng *et al.* [26] investigate via simulation a method of dynamic warp subdivision to hide branch and memory latency divergence. Similarly to our paper, they characterize the performance impact of variations in cache miss latency and other microarchitectural parameters; however, they primarily apply these modifications to their proposed dynamic warp hardware modification. Lee and Wu [22] also use GPGPU-Sim to examine program behavior in terms of stall cycle distribution. They focus on the Rodinia benchmark suite, examine fewer aspects of microarchitectural performance, and do not attempt to delineate the impact of irregularity. To the best of our knowledge, ours is the first use of a cycle-accurate simulator to characterize the performance and bottlenecks specifically of irregular CUDA codes.

## VI. Conclusions

This paper presents a microarchitectural workload characterization focusing on irregular GPU codes. We study the impact of control flow and memory access irregularity on several performance aspects, analyze how this behavior differs from regular GPU programs, and characterize the sensitivity of irregular code to cache and DRAM latency and bandwidth as well as cache size. We additionally connect source code to particular microarchitectural performance characteristics.

As expected, even extensively hand-optimized graph and tree algorithms achieve lower IPCs than regular codes. In general, they exhibit greater performance loss due to load imbalance, branch divergence, and uncoalesced memory accesses resulting from the unpredictable nature of their control flow and memory access patterns. This general trend is not always true, though. BH, for example, builds and operates on an irregular data structure (an octree) but, due to targeted code optimizations, displays less memory irregularity than FPC, which is not based on a dynamic data structure.

Interestingly, for these (presumably comprehensively optimized) irregular codes, load imbalance, branch divergence, and the overhead of synchronization and atomic operations limited the performance less than we expected. Memory-related slowdown is the single biggest factor limiting the performance of irregular applications, even those that have been demonstrated to possess ample parallelism [20], because irregular memory access patterns appear to be difficult to regularize and coalesce.

We find that, for our sizeable inputs, reducing the L2 latency and bandwidth is more important than decreasing the DRAM latency and bandwidth to improve the performance of programs with irregular memory accesses. Additionally, we observe that hardware strategies designed to reduce the penalty associated with uncoalesced memory accesses, including increasing in-core miss-handling resources, are unlikely to have a significant effect without corresponding improvements in memory bandwidth and latency.

## ACKNOWLEDGMENT

This work was supported by the National Science Foundation Graduate Research Fellowship Program under grant 1144466, as well as NSF grants 1141022, 1217231, and 1438963 and grants and gifts from NVIDIA Corporation.

## REFERENCES

[1] Alfred Aho, Ravi Sethi, and Jeffrey D. Ulman. *Compilers: principles, techniques, and tools*. Addison Wesley, 1986.

[2] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. *Proc. of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, 2009.

[3] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4):446-449, 1986.

[4] Emily Blem, Matthew Sinclair, Karthikeyan Sankaralingam. Challenge Benchmarks That Must be Conquered to Sustain the GPU Revolution. *Proc. of the 4th Annual Workshop on Emerging Applications and Many-Core Architecture*, 2011.

[5] Braunstein, M. Mezard, and R. Zecchina. Survey Propagation: An Algorithm for Satisfiability. *Random Structures and Algorithms*, 27:201-226, 2005.

[6] Martin Burtscher and Keshav Pingali. An efficient CUDA implementation of the tree-based Barnes Hut n-body algorithm. *GPU Computing Gems Emerald Edition*, pp. 75–92. Morgan Kaufmann, 2011.

[7] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A Quantitative Study of Irregular Programs on GPUs. *Proc. of the 2012 IEEE International Symposium on Workload Characterization*, pp. 141-151, 2012.

[8] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. Pannotia: Understanding Irregular GPGPU Graph Applications. *Proc. of the 2013 IEEE International Symposium on Workload Characterization*, 2013.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributing Computing*, 68:1370–1380, 2008.

[10] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, Kevin Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. *Proc. of the 2010 IEEE International Symposium on Workload Characterization*, pp. 1-11, 2010.

[11] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. *Proc. of the Symposium on Computational Geometry*, 1993.

[12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, McGraw Hill, 2001.

[13] CUB, http://nvlabs.github.io/cub/, 2014.

[14] Fermi. http://www.nvidia.com/content/PDF/fermi white papers/NVIDIA Fermi Compute Architecture Whitepaper.pdf, 2010.

[15] Nilanjan Goswami, Ramkumar Shankar, Madhura Joshi, and Tao Li. Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications. *Proc. of the 2010 IEEE International Symposium on Workload Characterization*, pp. 1–10, 2010.

[16] GPGPU-Sim 3.x Manual. http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual, 2012.

[17] Kirsten Hildrum and Philip S. Yu. Focused Community Discovery. *Proc. of the International Conference on Data Mining*, 2005.

[18] Song Huang, Shucai Xiao, and Wu chun Feng. On the energy efficiency of graphics processing units for scientific computing. *Proc. of the 23rd IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, 2009.

[19] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A Characterization and Analysis of PTX Kernels. *Proc. of the 2009 IEEE International Symposium on Workload Characterization*, pp. 3–12, 2009.

[20] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, and Keshav Pingali. How Much Parallelism is There in Irregular Applications? *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pp. 3-14, 2009.

[21] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala and L. Paul Chew. Optimistic Parallelism Requires Abstractions. *Proc. of the ACM Conference on Programming Languages Design and Implementation*, pp. 211-222, 2007.

[22] Shin-Ying Lee and Carole-Jean Wu. Characterizing the Latency Hiding Ability of GPUs. *Proc. of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 145-146, 2014.

[23] LonestarGPU, http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu

[24] Ryan Meltzer, Chi Zeng, and Cris Cecka. Micro-benchmarking the C2070 (poster). *GPU Technology Conference*, 2013.

[25] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. *Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 107–116, 2012.

[26] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *Proc. of the 37th International Symposium on Computer Architecture*, pp. 235-246, 2010.

[27] Duane G. Merrill, Michael Garland, and Andrew S. Grimshaw. Scalable GPU Graph Traversal. *Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012

[28] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph Algorithms on GPUs. *Proc. of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[29] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Atomic-free Irregular Computations on GPUs. *6th Workshop on General Purpose Processing on Graphics Processing Units*, 2013.

[30] Molly A. O'Neil and Martin Burtscher. Floating-point data compression at 75 Gb/s on a GPU. *Proc. of the 4th Workshop on General Purpose Processing on Graphics Processing Units*, pp. 7:1–7, 2011.

[31] Molly A. O'Neil, Dan Tamir, and Martin Burtscher. A Parallel GPU Version of the Traveling Salesman Problem. *Proc. of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 348–353, 2011.

[32] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron Lefohn, and Timothy J. Purcell. A survey of general purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[33] Victor Podlozhnyuk and Mark Harris. Monte Carlo Option Pricing. http://docs.nvidia.com/cuda/samples/4_Finance/MonteCarloMultiGPU/doc/MonteCarlo.pdf. 2012.

[34] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. *Proc. of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 235-246, 2010.

[35] Haicheng Wu, Gregory Diamos, Si Li, and Sudhakar Yalamanchili. Characterization and Transformation of Unstructured Control Flow in GPU Applications. *Proc. of the 1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, 2011.