A Quantitative Study of Irregular Programs on GPUs

Martin Burtscher
Texas State University
San Marcos, Texas, USA
Email: burtscher@txstate.edu

Rupesh Nasre
The University of Texas
Austin, Texas, USA
Email: nasre@ices.utexas.edu

Keshav Pingali
The University of Texas
Austin, Texas, USA
Email: pingali@cs.utexas.edu

***Abstract*—GPUs have been used to accelerate many regular applications and, more recently, irregular applications in which the control flow and memory access patterns are data-dependent and statically unpredictable. This paper defines two measures of irregularity called *control-flow irregularity* and *memory-access irregularity*, and investigates, using performance-counter measurements, how irregular GPU kernels differ from regular kernels with respect to these measures. For a suite of 13 benchmarks, we find that (i) irregularity at the warp level varies widely, (ii) control-flow irregularity and memory-access irregularity are largely independent of each other, and (iii) most kernels, including regular ones, exhibit some irregularity. A program's irregularity can change between different inputs, systems, and arithmetic precision but generally stays in a specific region of the irregularity space. Whereas some highly tuned implementations of irregular algorithms exhibit little irregularity, trading off extra irregularity for better locality or less work can improve overall performance.**

I. Introduction

Recent years have seen a surge of interest in the use of graphics processing units (GPUs) as general-purpose computing accelerators [28]. For programs that map well to GPU hardware, GPUs offer a substantial advantage over multicore CPUs in terms of performance, performance per dollar, and performance per transistor [38]. GPUs also outperform CPUs in energy efficiency on some applications [25]. Due to these benefits, GPUs are appearing as accelerators in many systems.

It is well-known that GPUs are very effective for exploiting parallelism in *regular* programs that (i) operate on large vectors or matrices, and (ii) access them in statically predictable ways. These codes often have high computational demands, exhibit extensive data parallelism, access memory in a streaming fashion, and require little synchronization [30]. A large number of algorithms from important application areas fit these criteria, including algorithms used in fields ranging from fluid dynamics [21] to computational finance [41]. There exists a broad base of knowledge on the efficient parallelization of these algorithms [18], and their GPU implementations can be tens of times faster than tuned parallel CPU versions [10].

However, many problem domains employ algorithms that build, traverse, and update *irregular* data structures such as trees, graphs, and priority queues. Irregular programs can be found in domains like *n*-body simulation [5], data mining [44], decisions problems that use Boolean satisfiability [6], optimization theory [12], social networks [23], system modeling [39], compilers [1], discrete-event simulation [35], and meshing [11]. They are more difficult to parallelize and more challenging to map to GPUs than regular programs.

Yet, several efficient GPU implementations of irregular algorithms have been published, demonstrating that GPUs are capable of accelerating at least some irregular codes relative to multicore CPUs [7], [32], [34]. However, little is known about the behavior of irregular GPU codes. For instance, we do not have clear answers to the following questions. The literature classifies applications as either regular or irregular, but does irregularity really manifest itself as a binary property? How is the irregularity behavior of an application influenced by its input, if at all? Although it seems like irregularity is bad for GPUs, does an increase in irregularity necessarily degrade performance or might it help in certain cases? The answers to these and related questions provide knowledge that is not only essential to understand application behavior but also to extract general insights and optimization techniques that can guide others in their efforts to accelerate irregular codes. This paper contributes as follows towards this goal.

- We present the first comprehensive workload characterization of a suite of real-world irregular GPU applications.
- We quantify two types of irregularity that are important to GPUs: control-flow and memory-access irregularity. Using our metrics, we illustrate that programs exhibit varying but consistent degrees of irregularity.
- We explore the input sensitivity of irregular kernels, *i.e.*, how the irregularity changes with the program input. We observe that a kernel may be input oblivious, input-type dependent, or input dependent.
- We investigate the effect of code optimizations on the characteristics of irregular applications. Our results indicate that performance improving optimizations, in certain cases, increase the amount of irregularity.

The rest of the paper is organized as follows. Section II reviews the concept of irregularity in programs, modern GPU hardware and the CUDA programming model. Section III defines and explains our metrics. Section IV presents the evaluation methodology and describes the applications we study. Section V discusses and analyzes the results. Section VI summarizes prior work. Section VII presents our conclusions.

II. Background

In this section, we first explain what we mean by irregularity in an application and then briefly introduce the GPU architecture and CUDA programming model. Although we focus on Fermi-based GPUs, the concepts in this paper should be applicable to other similar architectures.

A. *Regular versus irregular code*

The terms 'regular' and 'irregular' stem from the compiler literature. In regular code, control flow and memory references are not data dependent. Matrix vector multiplication is a good example. Knowing only the source code, the input size, and the starting addresses of the matrix and vectors, but without knowing any of the matrix or vector elements, we can predict the program behavior on an in-order processor, *i.e.*, the memory reference stream as well as the taken/not taken decisions of the conditional branch instructions.

In irregular code, both control flow and memory addresses may be data dependent. The input values determine the program's runtime behavior, which therefore cannot be statically predicted. For example, in a binary-search-tree implementation, the values and the order in which they are processed affect the control flow and memory references. Processing the values in sorted order will generate a tree with only right children whereas the reverse order will generate a tree with only left children, thus exercising a different control flow path. Even with unsorted inputs, the order of the values determines the shape of the tree as well as the order in which the tree is built, thus affecting the memory reference stream.

Graph-based applications in particular tend to be irregular. Their memory-access patterns are generally data dependent because the connectivity of the graph and the values on nodes and edges determine which graph elements are accessed by a computation, but the connectivity and values are unknown before the input graph is available and may change dynamically. Control flow is usually irregular for the same reasons.

B. *GPU and CUDA programming*

Fermi-based GPUs [15] contain up to 512 processing elements (PEs). Sets of 32 tightly coupled PEs form a streaming multiprocessor (SM). Whereas CUDA makes it appear as if each PE in an SM can run an independent thread of instructions, all 32 PEs must either execute the same instruction in the same clock cycle or some PEs have to wait. Internally, the PEs in each SM execute vector instructions that conditionally operate on 32 data items. A set of 32 threads that run together in this fashion is called a warp.

Warps in which not all threads can execute the same instruction are automatically subdivided by the hardware into sets of threads such that all threads in a set execute the same instruction. The sets are then serially executed until they re-converge. As a consequence, it is very important for performance to avoid thread divergence, *i.e.*, situations where not all threads in a warp follow the same control flow. This is how control-flow irregularity can hurt performance.

The memory subsystem is also optimized for warp-based processing. If the threads in a warp concurrently access words in main memory that lie in the same aligned 128-byte segment, the hardware merges the 32 reads or writes into one coalesced memory transaction that is as fast as accessing a single word. If multiple 128-byte segments are touched, the hardware has to perform correspondingly-many memory transactions, one after another. Thus, coalesced memory accesses are crucial to achieve a high memory bandwidth. This is the main reason why memory-access irregularity can lower performance.

The PEs within an SM share a pool of threads called thread block, an incoherent L1 data cache, and a software-controlled scratch pad called shared memory. A warp can simultaneously access 32 words in the shared memory as long as the words reside in different banks or all accesses within a bank request the same word. If more than one word is touched in a bank, a bank conflict occurs, which lowers performance. The hardware resolves these conflicts by sequentially accessing the conflicting words until all the requested data have been read or written. Bank conflicts are another reason for why memory-access irregularity may reduce performance.

The SMs operate largely independently. They can only communicate through global memory (DRAM). Thus, synchronization between SMs must be accomplished using operations on global memory locations. The PEs are fed with warps for execution from the thread pool in multithreading style to hide latencies. Because the PEs do not support out-of-order execution within threads but can arbitrarily interleave warps, it is important to have a large number of warps running in parallel to hide latencies.

III. GPU-relevant Metrics of Irregularity

In addition to locality, which affects the performance of both GPUs and CPUs, GPU performance is also heavily affected by divergent branches and memory operations that cause uncoalesced accesses or bank conflicts. Since locality is comparatively well understood, we focus on the other two factors in this paper. We use the following performance-counter-based metrics to express the degree of control-flow and memory-access irregularity.

$$\text{control-flow irregularity (CFI)} = \frac{\text{divergent\_branches}}{\text{executed\_instructions}}$$

$$\text{memory-access irregularity (MAI)} = \frac{\text{replayed\_instructions}}{\text{issued\_instructions}}$$

CFI is the fraction of the executed instructions that are divergent branches. MAI is the fraction of the issued instructions that are replayed. Both metrics range between 0% and 100%, with higher values representing more irregularity. Since the number of executed branches in typical applications is only a small fraction of the total number of instructions, and the number of divergent branches is a subset of all branches, CFI is usually low. For our benchmarks, it is always less than 4.1% (*cf.* Section V). A useful property of the two metrics is that they are independent of the runtime. Additionally, our metrics do not classify a program as regular or irregular. Rather, they measure the degree of irregularity along two dimensions.

Whenever there is either a bank conflict or a warp accesses more than one 128-byte block in memory, the corresponding load or store instruction is replayed by the hardware until all the requested data have been read or written. Hence, we can use the number of replayed instructions to capture both bank conflicts and uncoalesced accesses.

The denominators in the metrics include all instructions and not just branches or loads and stores to account for the overall frequency of divergence, bank conflicts, and uncoalesced accesses. Note that both metrics measure irregularity at the *warp level* as divergence across warps is not a problem for current GPUs and memory accesses can only be coalesced within a warp. Inter-warp divergence might affect the performance of the instruction cache, but almost all GPU kernels we are aware of fit entirely into the cache. Obviously, poor data locality between warps affects the performance of the data caches, but this is not the focus of our paper.

IV. Experimental Methodology

A. Hardware characteristics

We perform our experiments on a 1.45 GHz Quadro 6000 GPU with 6 GB of memory and 448 cores in 14 SMs. This Fermi-based GPU has 64 KB of fast memory per SM that is split between the L1 data cache and the shared memory. All SMs share an L2 cache of 768 KB. We compiled the CUDA programs (see Section IV-B) using *nvcc* v4.2 with the *-O3* and *-arch=sm_20* flags.

We use the Compute Visual Profiler v4.0.7 for collecting the performance-counter data from which we compute our metrics. Note that, in Fermi-based GPUs, only one SM is equipped with performance-counter hardware. Hence, we can only measure information about the thread blocks that execute on that SM. However, we believe that these thread blocks exhibit reasonably average behavior (*cf.* Section V-D), even in our highly irregular kernels. Moreover, our metrics are ratios and thus independent of absolute values, making them somewhat immune to differences in the load balance between the SMs. Finally, only a limited number of events can be counted concurrently, requiring the profiler to perform multiple runs of an application and to combine the results from these runs, which can lead to inconsistencies. We try to alleviate any such issues by using long-running inputs when possible.

For each application, we study the performance of the main kernel, which is usually the most time-consuming kernel. For the Barnes-Hut application, we consider two important kernels. For applications that invoke the main kernel multiple times, we accumulate the profile information from each invocation.

B. Applications

We characterize the following set of irregular programs. Their properties (lines of code using `wc -l`, the number of kernels, and the primary input) are listed in Table I.

B/M	LOC	#K	Input
BFS	1141	5	USA road network (23 M nodes, 58 M edges)
BH	1069	9	5 M bodies, 2 time steps
DC	674	1	msg_sp dataset, 28 blocks, 24 warps/block
DMR	958	4	10 M triangles
PTA	3494	40	vim (246944 pointers, 108271 constraints)
SP	445	3	2 M literals, 8.4 M clauses, 3 literals per clause
SSSP	614	2	USA road network (23 M nodes, 58 M edges)
TSP	454	3	kroE100 (100 cities, 200 K climbers)
BS	437	1	8 M options
HG	685	4	256 bins, 16 M random values
MC	1271	2	256 options
MM	499	3	640×1280
NB	2413	1	300 K bodies, 10 time steps

TABLE I: Application and input characteristics: B/M = benchmark, LOC = lines of code, #K = number of static kernels

- Breadth-First Search (BFS): This is a classic graph-traversal algorithm [34]. It involves labeling each node in the input graph depending upon its distance from a designated source node assuming unit edge weights. Since the distance of a node and its connectivity depend on the input graph, this algorithm exhibits irregular behavior.
- Barnes Hut (BH): This benchmark simulates the gravitational forces acting on a star cluster using the Barnes-Hut *n*-body algorithm [5], [7]. The positions and velocities of the *n* stars are initialized according to the empirical Plummer model. The program calculates the motion of each star through space for a number of time steps. In each step, the code hierarchically decomposes the space around the bodies into successively smaller volumes, called cells, and computes summary information for the bodies contained inside each cell, allowing the algorithm to quickly approximate the forces that the *n* bodies induce upon each other. The hierarchical decomposition is recorded in an octree. We investigate the tree-building kernel and the force-calculation kernel from this application, both of which are irregular because they build and repeatedly traverse an unbalanced tree, respectively.
- Data Compression (DC): This code implements a lossless compression algorithm for double-precision floating-point data [37]. It decomposes the input into chunks and processes each chunk in parallel. Because the chunks are read and written sequentially, this code does not perform any uncoalesced memory accesses. However, it does suffer from bank conflicts and especially thread divergence, both of which are caused by data dependent behavior, *i.e.*, how well each word can be compressed.
- Delaunay Mesh Refinement (DMR): This is a mesh-refinement algorithm from computational geometry [11], [29]. It works on a triangulated input mesh in which some triangles do not conform to certain quality constraints. The algorithm iteratively transforms such ‘bad’ triangles into ‘good’ triangles by recreating the neighborhood around each bad triangle. The newly created neighborhoods may contain new bad triangles, which are processed in a similar manner. The algorithm provably terminates with a mesh containing only good triangles. Since DMR deals with both deletions and additions of triangles, whose placement depends entirely on the input,

the control flow and the memory accesses are irregular.

- Points-to Analysis (PTA): This is Andersen's flow-insensitive, context-insensitive points-to analysis as used in compilers like GCC and LLVM [32]. It employs a fixed-point algorithm that operates on a dynamically growing constraint-graph in which directed edges are added depending upon the input points-to constraints. Since the graph structure is data-dependent and cannot be statically predicted, the algorithm is irregular.
- Survey Propagation (SP): Survey Propagation is a heuristic SAT-solver based on Bayesian inference [6]. The algorithm represents the Boolean formula as a factor graph, *i.e.*, a bipartite graph with variables on one side and clauses on the other. An edge connects a variable to a clause if the variable participates in the clause. The edge is given a value of -1 if the literal in the clause is negated and +1 otherwise. The general strategy of SP is to iteratively update each variable with the likelihood that it should be assigned a truth value of *true* or *false*. The memory access pattern of the algorithm is irregular as a literal may appear in any clause (depending upon the input), which cannot be statically predicted. The control-flow is irregular since the nodes processed by different threads may have varying degrees.
- Single-Source Shortest Paths (SSSP): This is another classic graph algorithm to compute the shortest path of each node from a designated source node in a directed graph with weighted edges [12]. We use an implementation of the Bellman-Ford algorithm. Similar to BFS, the shortest distance of a node depends upon the input graph and the memory access pattern is quite irregular.
- Traveling Salesman Problem (TSP): This is a classic NP-hard graph problem to find the shortest tour in a complete graph visiting all nodes exactly once and returning to the start node. The code implements an iterative hill climbing algorithm with random restarts for determining high-quality solutions (which may not be optimal) to the traveling salesman problem [36]. This implementation is semi-irregular as only the memory access pattern is data dependent.

To improve our understanding of irregularity, we also investigate, compare, and contrast the following codes from the CUDA SDK v4.1.15, most of which are considered regular.

- Black-Scholes (BS): This program evaluates fair call and put prices for a given set of options using the Black-Scholes formula. It works on a set of arrays containing floating-point numbers and is highly regular.
- Histogram (HG): This program performs 256-bin histogram calculations of arbitrary-sized 8-bit element data arrays. The array accesses are data dependent.
- Monte Carlo (MC): This program evaluates the fair call price for a given set of European options using the Monte Carlo approach. It works on an input array and exhibits quite regular behavior.
- Matrix Multiplication (MM): This is a classic highly regular matrix multiplication program.
- N-Body (NB): Similar to BH, this is an *n*-body program that simulates the motion of stars. It performs precise all-to-all force calculations, making it regular. In contrast, BH performs approximate force calculations and achieves an asymptotically better time complexity.

C. *Input characteristics*

The primary input used for each application is listed in the last column of Table I. The effect of different inputs on the same benchmark is discussed in Section V-B. For each benchmark, we tried to choose inputs that result in substantial kernel execution times.

V. Results and Analysis

In this section, we present and analyze our observations about the irregularity exhibited by various CUDA kernels. We first study the placement of our benchmarks in the irregularity space. Then, we investigate the effect of different program inputs. Next, we assess the influence of code optimizations and arithmetic precision on the amount of irregularity. We also examine the variability of the results between different runs on the same GPU and runs on different GPUs. Finally, we discuss other performance-counter results.

A. *Amount of irregularity*

Figure 1 shows a scatter plot of our programs where the x-axis represents the memory-access irregularity (MAI) and the y-axis represents the control-flow irregularity (CFI). Each point in the plot corresponds to one kernel. Recall from Section III that CFI is usually small. For our benchmark suite, we found it to always be less than 4.1%.

We observe that the benchmarks are quite scattered across the irregularity space. Most of the programs cannot strictly be classified as regular or irregular. Instead, each program has a certain *amount* of exhibited irregularity. For instance, HG and SSSP have high irregularity in both dimensions. In contrast, kernels like NB and BH (force) exhibit almost no warp-based irregularity (they are situated near the origin in Figure 1). Benchmarks like DC and, to a lesser extent, BFS

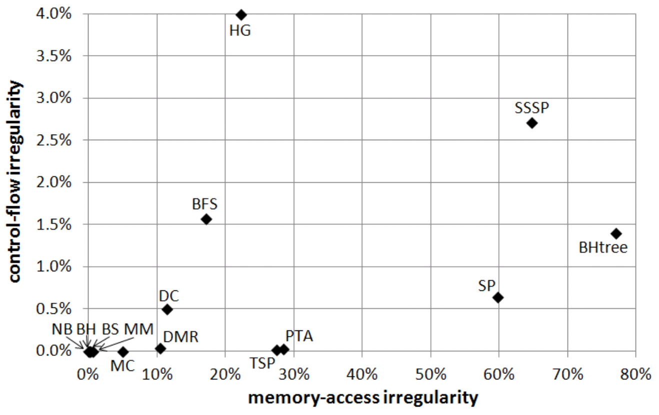

Fig. 1: Placement of kernels in the irregularity space

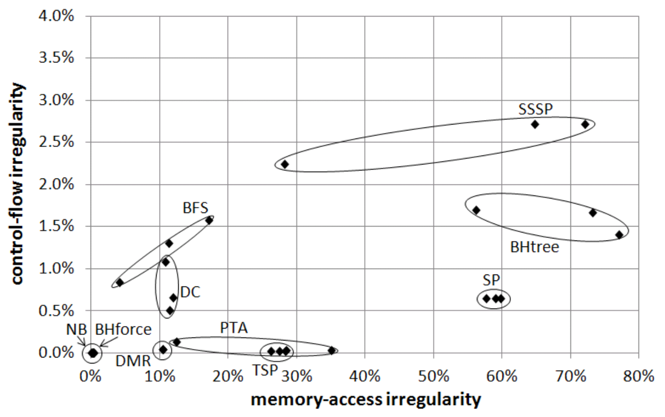

Fig. 2: Input sensitivity of various kernels (clusters are hand drawn to improve readability)

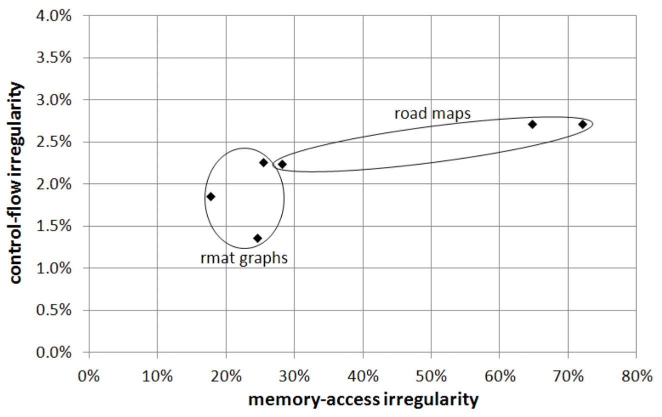

Fig. 3: Input-type sensitivity of SSSP

have a moderate amount of irregularity along both axes and cannot be categorized as highly regular or highly irregular.

The two kinds of irregularity appear to be fairly independent of each other. This is evident by the position of benchmarks like TSP and PTA, which have high memory-access irregularity but essentially no control-flow irregularity. The opposite, while theoretically possible, does not occur in our suite. Even HG exhibits 5.5 times as much MAI as it does CFI. Apparently, regular control flow is not correlated with a specific amount of memory-access irregularity, but irregular control flow generally implies irregular memory accesses.

Four of our five 'naturally' array-based CUDA SDK codes are, as expected, quite regular. NB, BS, and MM exhibit essentially no irregularity. MC has no CFI but a noticeable amount of MAI (5%). Interestingly, HG has by far the highest CFI of any of our benchmarks (4%) and also a substantial amount of MAI (22%). Clearly, not all array-based programs are regular (though many are). For example, the order in which the bins in a histogram are updated and how often each bin is updated are input dependent, making HG highly irregular.

B. Input sensitivity

This subsection assesses how the irregularity exhibited by an application depends on the program input. Since the degree of irregularity can depend on the problem size as well as on the actual values of the input, it is difficult to do this assessment in an application-independent way. We focus on studying how irregularity changes as a function of input size. Figure 2 displays the scatter plot for most of our benchmarks for different input sizes. Each presented kernel is profiled with three inputs: small, medium, and large (see column titled *input* in Table II for details on the three inputs).

The first observation is that, for many applications, the irregularity does not change drastically for different input sizes. By definition, irregularity in irregular applications is data dependent. Nevertheless, different inputs tend to cluster in the same region of the irregularity space. For NB, DMR, and the force-calculation kernel of BH (named BHforce), MAI and especially CFI essentially do not change between inputs. This is expected for highly regular applications (NB), regularized, *i.e.*, warp-based, implementations of irregular algorithms (BHforce), and some irregular kernels that process different-sized but otherwise similar data (DMR). TSP and SP exhibit a little variability. The remaining kernels, BFS, DC, PTA, SSSP, and BHtree, show larger changes in their irregularity when varying the input. For instance, the CFI and MAI of BFS differ between inputs by a factor of 1.9 and 4.2, respectively.

Interestingly, the MAI variability is generally higher than the CFI variability. The notable exception is DC, where different inputs mostly affect CFI. The reason for this unusual behavior is that the threads in a warp are unlikely to compress their current data word down to the same number of bytes, which affects the control flow of the loop that outputs the bytes but results in few bank conflicts because contiguous regions of shared memory are being written and there are no uncoalesced main memory accesses.

In general, the input sensitivity of an irregular application is difficult to predict. Consider, for example, BFS and SSSP, which we ran with the same inputs. Judging from the source code, we believe BFS exhibits a lower irregularity (along both axes) because it is more optimized than the SSSP implementation. Moreover, SSSP's high input sensitivity is an artifact of the input graph type. All three inputs are road networks, which tend to have a large diameter and a large variation in the node degrees, thus resulting in an unbalanced workload across the threads, which eventually leads to irregularity. To confirm this hypothesis, we reran SSSP on comparatively more uniform RMAT graphs and observed a noticeably lower irregularity as well as a lower variation thereof. Specifically, for RMAT graphs, the MAI ranges between 18% – 26% whereas for the US road networks, it ranges between 28% – 72%. These results are shown in Figure 3.

In summary, we identified the following three major types of input sensitivity.

1) *Input oblivious:* The irregularity behavior of these kernels remains largely constant for different inputs. The force-calculation kernel of BH is an example, which was explicitly coded in a warp-based manner to avoid thread

divergence and uncoalesced memory accesses [7].

2) *Input-type dependent:* The irregularity behavior of these applications varies less within a given type of input than across different input types (*e.g.*, road networks vs. RMAT graphs in SSSP).
3) *Input dependent:* The irregularity behavior of these applications changes considerably across inputs. BHtree is an example, which we ran on three different sized random inputs with similar general properties.

C. Effect of code optimizations and arithmetic precision

Irregularity is usually considered to hurt application performance. To study this, we examine how three code optimizations and a change from single- to double-precision arithmetic impact the irregularity of SSSP and the BH kernels.

a) BH optimizations: We investigate two largely identical implementations of the tree-building kernel in BH. The only difference is that one implementation explicitly records in memory the coordinates of each cell (internal tree node) whereas the more optimized version re-computes this information on the fly as it traverses the tree. Since computation is much cheaper than reading information from global memory, the optimized kernel runs up to 92% faster.

In spite of the resulting performance improvement, we can see in Figure 4 that this optimization actually increases the control-flow irregularity. This is unsurprising because the extra calculations include conditional statements that tend to be divergent. Nevertheless, this shows that it is possible to trade off more irregularity for better performance, thus demonstrating that control-flow irregularity is not the most important performance-determining factor on GPUs.

We observed a related effect on an optimization of the BH force-calculation kernel. Its runtime depends heavily on the order in which the bodies are processed. By sorting the bodies, the code runs over eight times faster, even when including the time to do the sort. However, because the kernel is written in a warp-centric fashion to avoid thread divergence and uncoalesced memory accesses, its CFI and MAI are near zero[1] regardless of whether the data are sorted or not (which is why we are not showing the results). This illustrates that performance can be independent of CFI and MAI.

b) SSSP optimization: Our SSSP implementation employs the traditional compressed sparse row (CSR) storage format to represent a sparse graph. However, this format does not exploit the connectivity between nodes. Since the algorithm pushes the newly computed distance information of a node to its outgoing neighbors, nodes that are logically close to each other in the graph should also be stored close to each other in memory to improve spatial locality. Greedily moving neighboring nodes closer in the data structure can improve the performance of the SSSP kernel by several factors.

Figure 4 presents the effect of this memory layout optimization on the irregularity behavior of SSSP for the three

[1]The BHforce kernel is nevertheless irregular as its inter-warp control flow and memory accesses are highly input dependent.

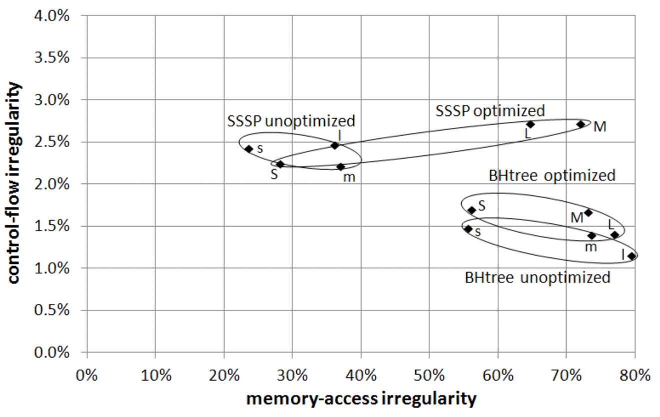

Fig. 4: Effect of code optimization on SSSP and the BHtree kernel; letters indicate small, medium, and large input

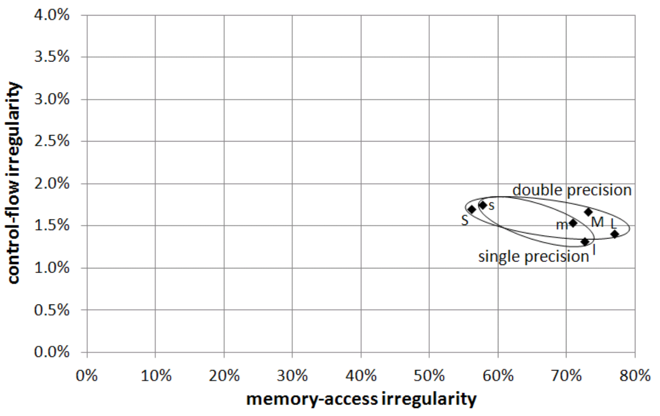

Fig. 5: Effect of floating-point precision on the BHtree kernel

road networks. For the small input, both CFI and MAI are not significantly affected by this layout change. For the medium input, the optimized layout increases the CFI by a small amount but doubles the MAI. For the large input, the optimized memory layout increases the CFI from 2.5% to 2.7% and the MAI from 36% to 65%. However, this increase in irregularity is compensated by improved spatial locality. In the absence of the memory layout optimization, nodes are sequentially assigned to threads. This enables warps to access the active nodes in a coalesced fashion, but the nodes' neighbors are scattered throughout memory, and accessing them results in many uncoalesced transactions that tend to miss in the L1 data cache. With the improved memory layout, the nodes are allocated based on their connectivity, which improves spatial locality for a thread when processing a connected subgraph. As a consequence, the initial node accesses are no longer coalesced, but visiting the neighbors now results in more cache hits due to the improved locality, yielding a net benefit in performance. Thus, memory-access irregularity and performance need not necessarily be negatively correlated.

c) BH arithmetic precision: Figure 5 illustrates the effect of changing the BH code from using single-precision to using double-precision floating-point data. Since double-precision values require two registers each and the number of registers per SM is fixed, the double-precision code uses fewer threads (and warps) per thread block than the single-precision version. Whereas we find that transitioning from single to double precision increases the tree-building kernel's CFI and MAI for the small input but decreases both metrics for the medium and large inputs, the change is quite small (3% – 8%) in all cases. This is an indication that a change in arithmetic precision will not substantially affect the irregularity of a program.

D. Variability

Figure 6 shows, on several of our kernels, by how much the irregularity varies between two runs on the same system (rhombuses ◆ vs. circles ○) and runs on different systems (rhombuses/circles vs. stars ×). The main differences between our two systems are that the second system runs in 32-bit mode, uses *nvcc* v4.0, and has a GTX 480 GPU with one more SM (480 CUDA cores), four times less global memory (1.5 GB), and a 3.5% slower clock speed (1.4 GHz) than our primary GPU.

The results are generally quite stable between runs on the same GPU. Only HG's irregularity shifts a little and MC's MAI shifts substantially. We believe non-determinism to be the reason for these changes in behavior. On the second GPU, the highly regular kernels NB and BHforce as well as the highly irregular kernels BHtree and HG exhibit almost exactly the same irregularity as they do on the primary GPU. TSP, DC, SP, MC, and SSSP yield noticeably different amounts of irregularity on the two GPUs. For SSSP, the most extreme case, the relative CFI difference is 28.7% and the relative MAI difference is 8.7%. Interestingly, MC, DC, and TSP mostly differ in MAI whereas SP and SSSP primarily differ in CFI. Overall, the measured irregularities are quite similar on the two GPUs, and both GPUs yield the same general characterization of each kernel. Based on these results, we believe our findings and conclusions to likely also be valid for other GPUs.

E. Other characteristics

Table II presents additional profile information for our benchmarks. The column-wise information lists the name, input, memory-access irregularity, control-flow irregularity, cumulative kernel runtime, instructions executed per active cycle, issued instructions as a percentage of executed instructions, occupancy, registers used per thread, active blocks per SM, active threads per SM, and shared memory per block.

Except for MM, the CUDA SDK kernels all have a high IPC. However, BHforce, DC, DMR, and TSP also have high IPCs even though, except for BHforce, they exhibit significant irregularity. Apparently, a reasonably amount of irregularity does not prevent a GPU from reaching a high IPC. TSP, with its substantial MAI of about 27%, reaches one of the highest IPCs we have measured. This is because TSP's working set fits into the shared memory, eliminating almost all main memory accesses, and the MAI is mostly due to bank conflicts.

The number of issued instructions is the number of executed instructions plus the number of replayed instructions (which

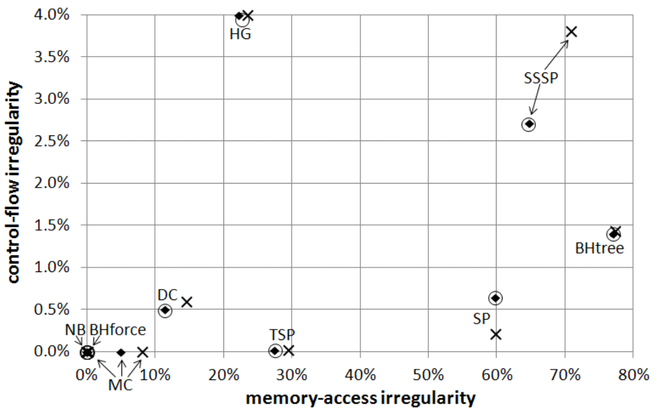

Fig. 6: Variability of irregularity between original run (rhombuses), another run on the same GPU (circles), and a run on a different system with a different GPU (stars)

we use to compute the MAI). Hence, the ratio of issued instructions over executed instructions (Issue%) is linked to the MAI through a simple relationship. As mentioned above, whenever a bank conflict occurs or a warp accesses more than one 128-byte block in memory, the corresponding load or store instruction is replayed until all the requested data have been accessed. This is why the number of issued instructions is often much larger than the number of executed instructions.

The occupancy measures the fraction of the number of threads (or warps) running in an SM relative to the theoretical maximum, *i.e.*, the number of active threads per SM divided by 1536. The occupancy does not have to be 1.0 to reach maximum application performance; in our experience, two thirds is often enough. As we can see, most of the irregular and all of the regular kernels reach this level.

To run more than 1024 threads per SM, at least two blocks must be running concurrently (active blocks per SM). Multiple active blocks can also be used to hide synchronization delays within thread blocks. All of the SDK kernels except MM use more than one active blocks per SM. Of the irregular codes, only BFS, DMR, and the BH kernels use multiple active blocks. However, multiple active blocks reduce the amount of available shared memory per block (last column of Table II). This is why SSSP and TSP can only run one block per SM. Moreover, if the register count per thread is too high, the full occupancy also cannot be reached, as is the case in NB, BFS, BHforce (particularly the double-precision version), DC, PTA, SP, and TSP. BS only runs 1024 active threads per SM because an SM cannot hold more than 8 active blocks. We are unsure why MM does not use smaller blocks to boost its occupancy.

VI. Related Work

Many irregular GPU implementations have been published, including single-source shortest paths (SSSP) and all-pairs shortest paths algorithms [20], breadth-first search (BFS) codes [14], [20], [34], algorithms for constructing kd-trees [48], Boruvka's minimum spanning tree algorithm [45], the MPM algorithm for maximum flow problems [42], the

Bench	Input	MAI%	CFI%	GPUTime (ms)	IPC	Issue%	Occu	Regs	Act blk per SM	Act thr per SM	Shmem (bytes)
BS		0.23	0.00	626.7	1.64	100.2	0.67	16	8	1024	0
HG	16 runs	22.25	4.08	67.5	1.54	128.6	1.00	18	8	1536	6144
MC	256 options	5.02	0.00	1.9	2.00	105.3	1.00	20	6	1536	2048
MM	960x640	0.57	0.00	881.0	1.00	100.6	0.67	20	1	1024	8192
NB	-n=60000	0.20	0.00	1178.6	1.91	100.2	0.67	29	4	1024	4096
NB	-n=115000	0.00	0.00	4335.4	1.95	100.0	0.67	29	4	1024	4096
NB	-n=300000	0.00	0.00	28556.9	1.96	100.0	0.67	29	4	1024	4096
BFS	fla	4.06	0.84	36.4	0.66	104.2	0.67	32	8	1024	6004
BFS	w	11.32	1.30	78.5	0.79	112.8	0.67	32	8	1024	6004
BFS	usa	17.13	1.58	228.6	0.78	120.7	0.67	32	8	1024	6004
BHforce	50K 50	0.32	0.00	2689.0	1.46	100.3	0.83	23	5	1280	3072
BHforce	500k 10	0.22	0.00	6292.4	1.59	100.2	0.83	23	5	1280	3072
BHforce	5M 2	0.15	0.00	13829.6	1.58	100.2	0.83	23	5	1280	3072
BHforce	50K 50 double	0.32	0.00	3781.8	1.46	100.3	0.50	36	3	768	4096
BHforce	500k 10 double	0.22	0.00	8947.2	1.59	100.2	0.50	36	3	768	4096
BHforce	5M 2 double	0.15	0.00	20573.3	1.58	100.2	0.50	36	3	768	4096
BHtree	50K 50	56.18	1.69	168.2	0.51	228.2	1.00	20	3	1536	0
BHtree	500k 10	73.25	1.66	144.8	0.78	373.9	1.00	20	3	1536	0
BHtree	5M 2	77.02	1.40	274.0	0.81	435.3	1.00	20	3	1536	0
BHtree	50K 50 norecalc	55.62	1.48	179.4	0.53	225.3	1.00	20	3	1536	0
BHtree	500k 10 norecalc	73.66	1.39	221.2	0.62	379.7	1.00	20	3	1536	0
BHtree	5M 2 norecalc	79.51	1.14	525.9	0.57	488.1	1.00	20	3	1536	0
DC	28 24 24 obs_error	11.93	0.66	1.0	1.59	113.5	0.50	24	1	768	6144
DC	28 24 32 msg_sweep3d	10.83	1.08	2.0	1.51	112.1	0.50	24	1	768	6144
DC	28 24 15 msg_sp	11.37	0.50	10.8	1.52	112.8	0.50	24	1	768	6144
DMR	2m	10.38	0.04	4163.5	1.47	111.6	1.00	19	6	1536	0
DMR	4m	10.39	0.04	3007.5	1.47	111.6	1.00	19	6	1536	0
DMR	10m	10.40	0.04	3391.0	1.47	111.6	1.00	19	6	1536	0
PTA	vim	28.45	0.03	4485.1	0.42	139.8	0.33	58	1	512	17920
PTA	gdb	35.00	0.03	4484.0	0.39	153.8	0.33	58	1	512	17920
PTA	svn	12.45	0.13	604.4	1.26	114.2	0.33	58	1	512	17920
SP	500k, 2.1m	57.70	0.65	3986.1	0.40	236.4	0.67	28	1	1024	0
SP	1m, 4.2m	59.09	0.65	8861.2	0.37	244.4	0.67	28	1	1024	0
SP	2m, 8.4m	59.74	0.65	18778.8	0.35	248.4	0.67	28	1	1024	0
SSSP	rmat5	17.78	1.85	0.1	0.19	121.6	0.67	19	1	1024	40960
SSSP	rmat6	25.50	2.26	0.2	0.21	134.2	0.67	19	1	1024	40960
SSSP	rmat10	24.61	1.36	1.8	0.76	132.7	0.67	19	1	1024	40960
SSSP	fla	28.27	2.24	91.4	0.78	139.4	0.67	19	1	1024	40960
SSSP	w	72.08	2.72	594.7	0.59	358.2	0.67	19	1	1024	40960
SSSP	usa	64.76	2.71	1732.6	0.60	283.8	0.67	19	1	1024	40960
SSSP	fla, noopti	23.62	2.42	921.3	0.90	130.9	0.67	19	1	1024	40960
SSSP	w, noopti	36.98	2.21	585.3	0.81	158.7	0.67	19	1	1024	40960
SSSP	usa, noopti	36.12	2.47	7067.1	0.82	156.6	0.67	19	1	1024	40960
TSP	kroE100 50000	26.18	0.01	1185.5	1.97	135.5	0.67	28	1	1024	48400
TSP	kroE100 100000	28.28	0.02	2175.9	1.96	139.4	0.67	28	1	1024	48400
TSP	kroE100 200000	27.50	0.02	4456.4	1.97	137.9	0.67	28	1	1024	48400

TABLE II: Profile data

Barnes-Hut (BH) *n*-body algorithm [7], 0-CFA analysis [40], Petri Net simulation [8], and Andersen-style points-to analysis (PTA) [32]. We include several of these codes in our study.

Che *et al.* present the Rodinia benchmark suite for heterogeneous computing infrastructures [9]. The current version contains BFS, another graph-traversal code, two applications that operate on unstructured grids, and otherwise mostly regular codes. Stratton *et al.* present the Parboil benchmark suite for throughput-computing architectures and compilers [43]. This suite includes programs like BFS, HG, MM, and several other regular codes. In contrast, the suite of applications we study contains primarily irregular codes.

There is some preexisting work on characterizing (mostly regular) GPU codes. Kerr *et al.* propose several metrics for analyzing the behavior of the CUDA SDK and two other codes [26] as well as an empirical model to predict the performance of GPU kernels [27]. Goswami *et al.* characterize the workload of several benchmarks from the CUDA SDK and the Parboil and Rodinia suites [19]. They propose a set of microarchitecture-agnostic properties and cluster the studied benchmarks using those properties to understand relative workload similarities. They also present a diversity analysis in the context of memory coalescing and branch divergence.

A number of papers examine control-flow and memory-access patterns in GPU codes and suggest optimizations to reduce irregularity. A few studies propose empirical models to predict the effect of control-flow divergence and the memory hierarchy on performance [2], [3], [13]. Bakhoda *et al.* study non-graphics kernels on a GPU simulator [4]. They find that reducing the number of concurrent threads can sometimes improve overall performance by reducing memory contention. Wu *et al.* study several benchmarks to identify the sources of control-flow irregularity [46]. They argue that GPU hardware provides only limited support for unstructured control flow and propose a compiler transformation to automatically convert unstructured control flow into structured control flow. Zhang *et al.* propose G-Streamline to eliminate control-flow and memory-access irregularities on the fly [47]. Hong *et al.* propose virtual warp sizing to speed up irregular GPU implementations of BFS and other codes [24]. Hetherington *et al.* illustrate that the presence of control-flow divergence does not necessarily render the corresponding application ineligible for achieving good performance on GPUs [22]. They show that, by optimizing the memory layout, the irregular Memcached application can obtain considerable speedup. There are also papers that propose hardware improvements to better support the execution of irregular codes on GPUs [16], [17], [31], [33].

To the best of our knowledge, ours is the first comprehensive characterization of irregular graph algorithms on GPUs.

VII. Conclusions

We present the first workload characterization of an entire suite of irregular GPU programs. We primarily characterize each program using two runtime-independent metrics we developed that quantify the control-flow irregularity (CFI) and the memory-access irregularity (MAI) at the warp level. Both metrics are derived from performance-counter measurements. We characterize and contrast 8 irregular and 5 regular codes. In some cases, we study multiple versions of an application.

There is no dichotomy between regular and irregular codes because the amount of irregularity varies widely between GPU kernels and because most kernels exhibit some degree of irregularity, including so-called ‘regular’ kernels. Hence, we believe it is misleading to categorize applications as either regular or irregular. Rather, an application should be characterized by the amount of irregularity it exhibits. The CFI and MAI metrics are intended for this purpose.

Irregularity is not necessarily bad for performance. Whereas many highly regular and some warp-based irregular kernels that perform well exhibit little irregularity, we found several code optimizations that do not change the irregularity or increase it, yet result in a speedup. This indicates that irregularity is not the most important performance-determining factor. On the contrary, it can be beneficial to trade off additional control-flow or memory-access irregularity for an improvement in locality or for a reduction in the amount of work, particularly memory accesses. Hence, we recommend this trade-off to be considered when implementing and tuning irregular GPU applications.

Although irregular codes are, by definition, data dependent, programs tend to occupy a relatively narrow range in the irregularity space, *i.e.*, different inputs yield similar degrees of irregularity, making a general kernel characterization by irregularity possible. Depending on the ‘size’ of this range, we identified three main categories of input sensitivity: input oblivious (little or no variability between inputs), input-type dependent (distinct variability between different types of inputs but not much variability between inputs of the same type), and input dependent (significant variability between inputs).

We also studied the variability of our results between multiple runs on the same and on different systems as well as the effect of arithmetic precision. We found the CFI and MAI to be very stable across runs and to vary somewhat between distinct GPUs and when changing from single to double precision. We therefore expect our conclusions to hold across a broad range of CUDA-capable GPUs and hope that our findings will increase the understanding of the behavior of irregular GPU applications.

VIII. Acknowledgments

This work was supported by NSF grants 0833162, 1062335, 1111766, and 1217231 as well as grants and equipment donations from NVIDIA, IBM, Intel and Qualcomm Corporations.

References

[1] A. Aho, R. Sethi, , and J. Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, 1986.

[2] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for GPU architectures. *SIGPLAN Notices*, 45(5):105–114, January 2010.

[3] Sara S. Baghsorkhi, Isaac Gelado, Matthieu Delahaye, and Wen-mei W. Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 23–34, New York, NY, USA, 2012. ACM.

[4] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[5] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4), December 1986.

[6] A. Braunstein, M. Mèzard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27(2):201–226, 2005.

[7] Martin Burtscher and Keshav Pingali. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In *GPU Computing Gems Emerald Edition*, pages 75–92. Morgan Kaufmann, 2011.

[8] Georgios Chalkidis, Masao Nagasaki, and Satoru Miyano. High Performance Hybrid Functional Petri Net Simulations of Biological Pathway Models on CUDA. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(6):1545–1556, November 2011.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[10] 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, October 2008.

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

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

[13] Zheng Cui, Yun Liang, Kyle Rupnow, and Deming Chen. An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2012.

[14] Yangdong (Steve) Deng, Bo David Wang, and Shuai Mu. Taming irregular EDA applications on GPUs. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 539–546, New York, NY, USA, 2009. ACM.

[15] Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010.

[16] Wilson W. L. Fung and Tor M. Aamodt. Thread block compaction for efficient SIMT control flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 25–36, Washington, DC, USA, 2011. IEEE Computer Society.

[17] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Gene Golub and Charles van Loan. *Matrix computations*. Johns Hopkins Press, 1996.

[19] Nilanjan Goswami, Ramkumar Shankar, Madhura Joshi, and Tao Li. Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC'07: Proceedings of the 14th international conference on High performance computing*, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] Mark Harris. Fast fluid dynamics simulation on the GPU. In *ACM SIGGRAPH 2005 Courses*, New York, NY, USA, 2005. ACM.

[22] T.H. Hetherington, T.G. Rogers, L. Hsu, M. O'Connor, and T.M. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 88–98, April 2012.

[23] Kirsten Hildrum and Philip S. Yu. Focused Community Discovery. In *International Conference on Data Mining*, 2005.

[24] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 267–276, New York, NY, USA, 2011. ACM.

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

[26] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of PTX kernels. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, pages 3–12, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. Modeling GPU-CPU workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 31–42, New York, NY, USA, 2010. ACM.

[28] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[29] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *SIGPLAN Notices (Proceedings of PLDI)*, 42(6):211–222, 2007.

[30] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28:39–55, 2008.

[31] Roman Malits, Evgeny Bolotin, Avinoam Kolodny, and Avi Mendelson. Exploring the limits of GPGPU scheduling in control flow bound applications. *ACM Transactions on Architecture and Code Optimization*, 8(4):29:1–29:22, January 2012.

[32] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 107–116, New York, NY, USA, 2012. ACM.

[33] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 235–246, New York, NY, USA, 2010. ACM.

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

[35] Jayadev Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[36] M. A. O'Neil, D. Tamir, and M. Burtscher. A Parallel GPU Version of the Traveling Salesman Problem. In *2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 348–353, 2011.

[37] Molly A. O'Neil and Martin Burtscher. Floating-point data compression at 75 Gb/s on a GPU. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 7:1–7:7, New York, NY, USA, 2011. ACM.

[38] 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.

[39] James L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[40] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 511–522, New York, NY, USA, 2011. ACM.

[41] Steven Solomon, Ruppa K. Thulasiram, and Parimala Thulasiraman. Option Pricing on the GPU. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, pages 289–296, Washington, DC, USA, 2010. IEEE Computer Society.

[42] Steven Solomon and Parimala Thulasiraman. Mapping the MPM maximum flow algorithm on GPUs. *Journal of Physics: Conference Series*, 256(1):012006, 2010.

[43] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois, at Urbana-Champaign, 2012.

[44] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar, editors. *Introduction to Data Mining*. Pearson Addison Wesley, 2005.

[45] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 167–171, New York, NY, USA, 2009. ACM.

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

[47] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–380, New York, NY, USA, 2011. ACM.

[48] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):1–11, 2008.