Fine-Grained Management of Thread Blocks for Irregular Applications

Jonathan Beaumont* Trevor Mudge*

*University of Michigan, Ann Arbor, MI
*{jbbeau, tnm}@umich.edu

***Abstract*—Co-locating threads with complementary resource usage is a key strategy for improving throughput in parallel machines such as GPUs. However, the proliferation of irregular algorithms which change execution behavior dynamically makes optimal thread placement impossible when done statically or by profiling kernels as a whole. In this work, we characterize the performance loss associated with current thread block scheduling policies in GPU architectures. We then demonstrate that an extension of these strategies incorporating dynamic performance metrics such as memory and functional unit utilization at the thread block level as well as preemptive thread block swapping can improve throughput. We show that performance on irregular algorithms can be improved by an average increase of 17.1% over static profiling methods and 12.9% over dynamic strategies with no changes to legacy software and minimal hardware extensions which increase SRAM storage area by less than 0.5%.**

***Index Terms*—GPUs, thread scheduling, resource management, preemption, irregular parallelism**

I. Introduction

There has been a significant trend in mapping an increased variety of workloads to data parallel architectures. We have seen numerous enhancements to GPU hardware by vendors to enable greater flexibility and programability [1] [2] [3] as well as the emergence of new architectures for specific domains, such as Google's Tensor Processing Unit [4] to accelerate machine learning algorithms. A prevalent challenge is the increasing amount of control and data irregularities present in many non-trivial algorithms [5], making it difficult to fully utilize hardware. Multi-kernel execution and virtualization of hardware across multiple users have been employed on GPUs to mitigate these problems [6], but it is difficult to scale these solutions in light of nondeterminism introduced by irregular algorithms.

In this work, we will explore specifically how multi-kernel techniques are hampered by irregular data parallelism, and present our design for an extension to thread block scheduling and placement that will circumvent these challenges. Our contributions are as follows:

- We identify and characterize the variability of resource usage within and across thread blocks when running irregular algorithms as a function of input density, resulting in distinct phases of computation
- We demonstrate that previous methods of resource allocation, both static and dynamic, are insufficient at accounting for this variability and result in interference between thread blocks
- We propose a solution which adds hardware to track and predict future resource usage for each thread block. This hardware is used to make more sophisticated decisions for allocating resources, as well as dynamically detecting phase shifts and reallocating resources when appropriate
- We show that our solution results in a 17% and 13% average increase in throughput over established static and dynamic scheduling strategies, respectively, while only increasing total SRAM requirements by less than 0.5%

II. Background

A. GPUs

NVIDIA's CUDA GPUs use a single instruction multiple thread (SIMT) programming model, in which the programmer writes code for a single *thread* that is replicated hundreds to thousands of times on the hardware to operate on large sets of data in parallel. The GPU hardware consists of several *streaming multiprocessors* (*SMs*), each containing a large register file to support many threads, a cache-hierarchy including a software managed scratch-pad (also called *shared memory*), and several multithreaded single instruction, multiple data (*SIMD*) pipelines.

Threads are grouped into *thread blocks* (*TBs*) by the programmer, which are guaranteed to run on the same SM and can cooperate through barrier instructions and a scratchpad called *shared memory*. Thread blocks are dispatched to SMs via the global *thread block scheduler* (*TBS*) as a *grid* when enough resources (namely register file and shared memory space) are available and are run asynchronously from one another until all threads within the thread block have completed. A programmer can indicate that independent kernels can be run concurrently by issuing them on different *streams*.

SMs do not employ branch prediction or out-of-order mechanisms, but instead rely on swapping between the many available threads on a cycle-by-cycle basis to hide arithmetic and memory latencies. Thread blocks are transparently divided into smaller bundles of threads called *warps*, which are executed in lockstep, simplifying fetch and dispatch logic. These considerations allow a much greater portion of a GPU's die area to be dedicated towards computation compared with scalar

978-1-5386-6648-7/19/$31.00 ©2019 IEEE
DOI 10.1109/ICCD46524.2019.00042

cores, and enable their high-energy efficiency and throughput capabilities [7].

While the SIMT programming model allows for the use of conditional statements within code which create the appearance of threads being able to run independently of one another, thread divergence (the occurrence of multiple distinct branch outcomes within a single warp) and memory access divergence (different cache lines being accessed within a single warp memory operation) results in lane operations to be serialized and underutilization. Programmers must take care to ensure that control-flow and memory-access irregularity are minimized as much as possible to ensure good performance.

B. Irregular Algorithms

Data-parallel architectures such as GPUs have been ideal platforms for accelerating algorithms which involve unconditionally applying repeated operations on logically contiguous data. Matrix-matrix multiplication is an archetypal application which can achieve over $100\times$ improvement in performance per Watt on a GPU over serial implementations, as data can be streamed very efficiently through its hundreds of available processing units with minimal control overhead [8].

While not to the same degree, a wider class of algorithms referred to as irregular data parallel algorithms (also called “amorphous data parallel algorithms” or simply “irregular algorithms”), has been demonstrated to show speedups of $2 - 10\times$ on parallel architectures [5]. These algorithms are still data parallel in the sense that operations are applied repeatedly across multiple pieces of data, but the control flow and memory access patterns are less regular and predictable. Many problems which can be represented as graphs, where operations are propagated through adjacent nodes, are irregular, since the memory layout is data dependent and may change throughout execution. While these irregularities make it difficult to fully utilize hardware, several techniques have been demonstrated to show practical speedups on parallel architectures [9] [10] [11].

```
__global__ void topo(Node *nodes, bool *done) {
  Node node = nodes[threadIdx];
  if( node.active() ) {
    node.process();
    *done = false;
  }
}

int main() {
  // ...
  while (!done) {
    done = true;
    topo<<<N>>>(nodes, &done);
  }
}
```

Fig. 1. Topology-driven implementation of an algorithm, which spawns N threads per iteration. Each thread checks whether its assigned node is active, and if so processes it. The ”process” method may activate neighboring nodes

Two common approaches towards designing these algorithms are topology-driven and data-driven implementations [12]. In both of these styles, the work to be done is represented by a set of connected nodes. Some subset of these nodes are active, indicating that they must be processed by applying an algorithm-specific operator on them. After processing each node, the neighboring nodes may or may not be activated, causing a propagation effect of iteratively processing active nodes until a steady state or some exit condition is reached. For example, in breadth-first search, the operator simply increments a distance variable by one, and all neighboring nodes are always activated.

Topology and data-driven implementations are different in how processing elements are assigned active nodes to process. In topology-driven implementations (an example is shown in figure 1), threads are generated to process each node regardless of whether there is useful work to be done. Such implementations are usually simple to design, but for data sets where the percentage of active nodes at any given time is low, this approach will be very inefficient, as few threads are performing meaningful work by executing the “process” method.

```
__global__ void data(Node *nodes, WL *wl) {
  while(idx = wl->pop()) {
    Node node = nodes[idx];
    node.process();
    for(i=0; i<node.num_neighbors; i++) {
      wl->push(node.neighbor(i));
    }
  }
}

int main() {
  // ...
  init<<<N>>>(nodes, wl);
  data<<<M>>>(nodes, wl);
}
```

Fig. 2. Data-driven implementation of an algorithm, which spawns M threads, each iteratively popping work assignments of a shared worklist and adding new work items back on

To address this, data-driven implementations (see figure 2) dispatch a fixed set of threads which persist throughout the application’s execution and are assigned nodes to process by a shared worklist maintained in software. Nodes are only added to this worklist when they are activated, so there are no idle “spin-loops” as there are in the topology-driven implementation.

Data-driven implementations are more algorithmically efficient as they avoid unnecessary work when processing inactive nodes. However, memory contention caused by accessing a shared worklist with atomic memory operations means such implementations are rarely used without aggressive software optimizations.

One such optimization is employing distributed worklists via thread-block partitions [12] [11]. This potentially sacrifices parallelism when load-balancing is needed but allows scaling to much larger workloads with low *density*, which we define as

$$D = \frac{2|E|}{|V|(|V|-1)}$$

where V is the total number of vertices or nodes, and E is the total number of edges or adjacencies between the nodes.

The work presented in this paper makes use of the LonestarGPU benchmark suite Version 3.0 [5] to study the behavior of irregular algorithms. The benchmark suite has provided many implementations of several key irregular algorithms, but as the primary concern is how computation will scale with the proliferation of large, sparse data sets, analysis of data-driven implementations is limited to those which use distributed worklists as part of their optimizations. Thus, the following benchmarks are used:

Breadth-first search (BFS) counts the minimum number of nodes between a specified source node and all other nodes. Starting with the source node, neighboring nodes are activated and updated with the current distance. There is a single kernel invocation which iterates until the graph is traversed, and the operator is very simple [13].

Delaunay mesh refinement (DMR) is provided with a mesh of nodes and iteratively modifies edges between them to resolve constraint violations. Because it modifies the underlying data structure, it is a "morph algorithm" [14], and invokes overhead since extra synchronization via barrier instructions is needed to prevent partially updated graphs from being read. It alternates invoking two separate kernels: one to modify the mesh, and the other to check for new violations.

Minimum spanning tree (MST) is another morph algorithm which finds a subset of the input graph that spans all nodes and has the smallest overall cost. It uses Boruvka's algorithm and alternates between two kernels to iteratively find the minimum weight edge coming from each component and then merge partners across those minimum edges. The kernels become more computationally intense as the graph is reduced.

Points-to-Analysis (PTA) determines the set of addresses a pointer variable in a program's source code can access given a set of constraints. The algorithm uses a pipeline of low-latency ($<$100 μs) kernels to iteratively propagate constrains to different nodes until a steady state is reached. [15]

Single-Source Shortest Paths (SSSP) calculates the shortest distance from a specified source node to all other nodes in a weighted graph, in a similar manner to BFS, but with slightly higher bandwidth needed to read the weights and higher computational intensity to process them.

Stochastic Gradient Descent (SGD) completes unknown entries in a supplied sparse matrix. It involves several dot product calculations between vectors and is the most computationally intensive workload considered.

III. Motivation

While previous works [16] [17] have recognized the problem of interference across thread blocks, none have considered the added difficulty when processing irregularly structured data and how resource usage of threads changes as a result. This is of paramount importance when considering the increased prevalence of data sets with non-uniform patterns [18]. Real world data sets often consist of graphs with clusters of highly-connected nodes dispersed across many sparsely-connected nodes. For example, figure 3 shows the distribution of different local densities in a graph representing which Netflix movies were enjoyed by which users [5]. While about three quarters of the graph's nodes are quite sparsely structured, having connections to between .001% and 1% of the other nodes, the remaining quarter of the nodes have a wide range of higher densities. This is an example of a "power-law" graph, which are becoming ever more prevalent in the era of big-data computing [19].

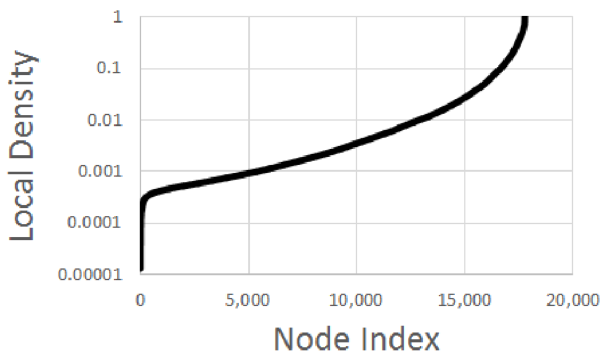

Fig. 3. Distribution of densities of a graph where nodes are sorted by number of connected neighbors. Local density of a given node is the number of connected neighbors for that node divided by the total number of nodes

Figure 4 shows two prominent effects this distribution has on the performance of algorithms processing such input data sets. Using an NVIDIA Tesla V100 GPU (the specifications of which are described in section V), we measured how the rate of stalls resulting from memory throttling (i.e. when the load-store-unit (LSU) is fully saturated and can't accept more requests) changes as the density of the data is increased across benchmarks from the LonestarGPU suite. Across all benchmarks, there is a decrease in the rate of these stalls as density increases. This is a consequence of higher data-reuse when processing dense data. As long as the working data set can be fit within the cache, tightly packed clusters of nodes will result in the compute operator processing the same nodes multiple times in quick succession, yielding a higher cache hit rate. Similarly, sparse segments of data will result in pointer-chasing with little data reuse and the LSU is more likely to be saturated with long latency cache misses. BFS and SSSP are both memory bound workloads with very simple compute operators, and accordingly show significant changes (a total of 78% and 93%, respectively) in the rate of stalls as the density and data reuse increases. On the other extreme, MST and SGD have much more computationally intense operators which hide the latencies associated with cache misses. The penalties associated with processing highly sparse data is correspondingly less than workloads with simple compute operators, with a 10% and 25% reduction of stalls in processing the densest data sets, respectively. DMR and PTA

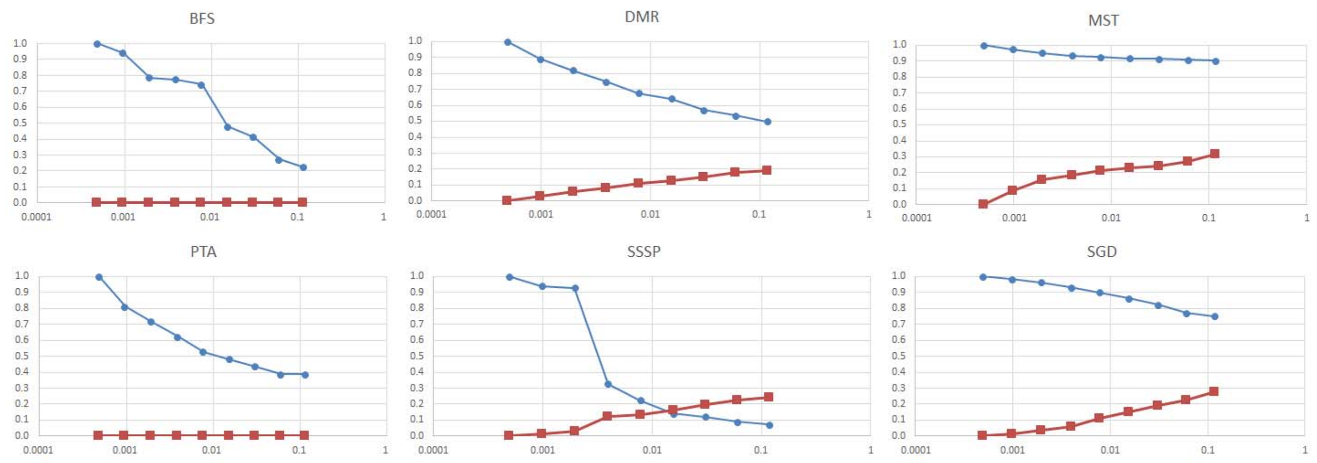

Fig. 4. How different metrics (y-axis) vary as a function of input density (x-axis) for each benchmark. The blue line is rate of stalls due to memory throttling, and the orange line is the increase in utilization of the floating point units. Both are normalized to the sparsest case studied

show moderate reductions in stalls when processing dense data (50% and 61%, respectively).

We also measured how the utilization of floating point units (FPUs) are affected by density. Both BFS and PTA do not contain any floating point code and are thus unaffected. DMR and SSSP show increased utilization (19% and 24%, respectively), but this is largely a side effect of the reduction of memory stalls; since less time is spent waiting for memory, a larger percentage of execution is spent processing data and utilization is increased. The amount of computation needed to process each node is largely unaffected. MST and SGD both have operators which grow in complexity as the density increases. For example, SGD performs numerous dot products on segments of the input data. As the number of non-zero elements within a row increases (i.e. as the density increases), more elements must be multiplied and summed together for each dot product, increasing the computational intensity of the operator. Accordingly, MST and SGD both see an increase in FPU utilization (31% and 28%, respectively) that is partially independent of the decrease in memory stalls. This fact will be significant in identifying benefits of tracking both metrics in tandem.

Because different thread blocks process independent data, there is little guarantee about how sparsity levels of the data correlate between active thread blocks. Over time, the threads processing these nodes may migrate to neighborhoods of different density, and correspondingly the resource usages of the containing thread blocks may also change. Accordingly, an optimal thread block scheduler should treat individual thread blocks within a given kernel differently depending on the properties of the data being processed. We make two observations that guide our design decisions for a finer granularity scheduling and resource allocation policy:

- The density of graphs changes smoothly as the graph is traversed. We observe that across the input data sets considered, 86% of 5-node diameter subraghs (i.e. the data processed by a thread across 5 iterations of the algorithm) see deviations of fewer than 10 neighbors per node between the most and least connected nodes in the subgraph.
- In both topology-driven and data-driven workloads which use distributed worklists, thread blocks with a given block index traverse graphs smoothly (that is, nodes accessed within in a short amount of time are logically near one another in the graph) even across separate kernel invocations. This is not the case with data-driven implementations which use a single shared worklist, as which particular subset of data accessed by a given thread block will be completely random across separate kernel invocations depending on the precise interleaving of other thread block's access to the worklist.

These observations suggest that we can predict how much of a particular resource a pending thread block is likely to use in the near future based on how that resource was used by previous instances of the same thread block (i.e. have the same block index). By tracking the resource usage via performance counters already present on modern GPUs, we can feed this information to the thread block scheduler and more accurately place thread blocks on the ideal SM.

However, this is not a general solution. Table I lists several properties of kernel iterations in the LonestarGPU benchmarks when processing a fixed-size data set. One of these is the average execution latency, or how long each kernel instance runs on average. DMR, MST, PTA, and SGD all invoke kernels several times across its execution, with the number of iterations depending on the size and structure of data. BFS and SSSP only invoke their respective kernels once, and as such their latencies make up most of the applications' run times.

The changing density patterns in the input data result in the emergence of "phases" where a thread block's resource usage changes over time. For the purposes of this work, we define a phase shift to have occurred when a given metric of interest (namely, the rate of memory throttle stalls or utilization of the FPU) averaged across a window of the last 100 μs deviates by

Application	Grid / Block size	Regs per thread	SMem per block (B)	Total context size (KB)	Preemption latency (μs)	Execution latency (μs)
BFS	240 / 128	32	0	16	3.0	56,000
DMR	112 / 512	32	0	64	11.8	9,080
MST	640 / 256	34	0	34	6.2	1,030
PTA	80 / 991	28	200	108	20.1	40
SSSP	160 / 256	32	0	32	5.9	23,000,000
SGD	900 / 16	26	68	2	0.3	10

TABLE I
AVERAGE ITERATION METRICS PER APPLICATION

more than 5% (choosing a smaller window size for the running average is unlikely to be of use, since, as we will explain later in this section, there will be no way to effectively respond to this variances on that time scale). The exact properties of these phases varies widely based on the benchmark as well as size and structure of the input data. We provide one example of the distribution of how long these phases last when executing the SGD benchmark in figure 5. There is not a clear upper bound on the length of these phases, but the majority last between 1 and 10 ms, with an average of about 5.4 ms. Across all benchmarks and several input data sets, the average phase length varies between 0.1 ms and 100 ms.

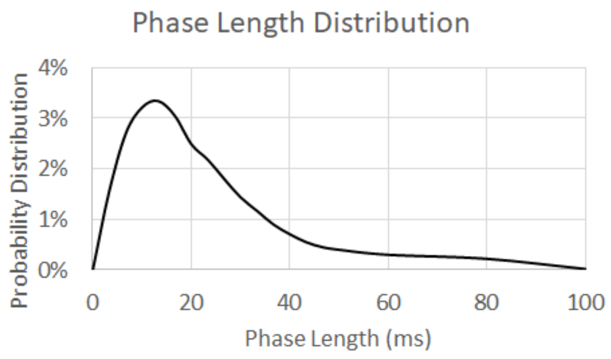

Fig. 5. Distribution of phase lengths when running the SGD benchmark

The key observation is that while half of the benchmarks (SGD, PTA, and to a lesser extent MST) have average latencies on the order of or significantly less than the typical phase lengths we observe and can therefore be rescheduled at a fine-granularity at the next invocation of that thread block, the other half (DMR and to a much larger degree BFS and SSSP) have average latencies significantly higher than typical phase lengths. Waiting to make allocation decisions at thread-block dispatch may potentially miss several phase changes entirely. Indeed, since BFS and SSSP only invoke their kernel once, we have no hope of adjusting the allotted resources based on phase changes at kernel dispatch.

Thus, the second part of an ideal resource allocation strategy is to detect phase changes in real time, preempt thread blocks which are not ideally placed, and relocate them to somewhere better suited.

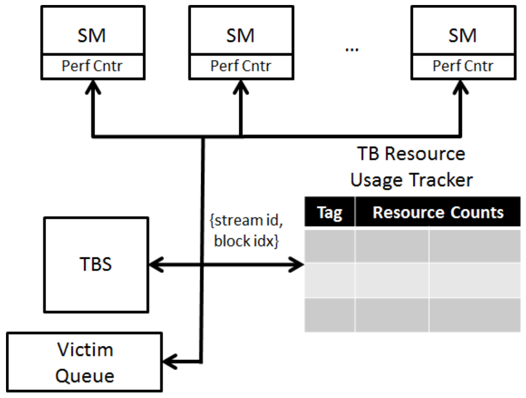

Fig. 6. Diagram of the proposed system architecture with the SMs, thread block scheduler, and the proposed victim queue and thread block resource usage tracker

IV. Design

We use all of the discussed observations to design a dynamic thread block scheduling and preempting infrastructure to better approximate an ideal solution. The architecture is summarized in figure 6. In order to account for resource usage variance across thread blocks, we propose the inclusion of a *thread block resource usage tracker (RUT)* which is implemented as an SRAM table in hardware. The RUT is indexed with both the stream ID and block index and stores an N bit unsigned integer for each utilization metric to be tracked by the TBS. The reason why the stream ID is used instead of the kernel ID is that applications frequently use a pipeline of kernels with identical dimensions to process data in several steps, where each corresponding thread block with the same block index processes the same subset of data and therefore exhibit similar resource usage trends.

In addition to resources such as shared memory and register file usage, the TBS will also estimate different resource usages through the RUT and use these estimates as additional constraints when placing thread blocks.

When allocating a thread block, the TBS will:

- access the RUT with the stream ID and block index to receive estimates on key resources
- find the first SM (using a rotating round robin policy) that has enough register file, shared memory, and supplemental resources to accommodate the current thread block

- if no SMs meeting that criteria are available, thread block dispatch is stalled until one becomes available

Periodically, each SM will check the resource usage of of each resident thread block using preexisting performance counters. If any tracked resource has deviated over a certain threshold since the previous check, the SM informs the TBS, which then updates the appropriate RUT entry. Each entry will also be updated when the given thread block terminates. Care must be taken to choose the right frequencies and parameters so that the system is not overrun with traffic from updates.

Whenever the RUT is updated by a still-running thread block, the TBS will reevaluate the resource usage of the SM. If the estimated resource usage has increased such that the SM is predicted to be oversubscribed, the SM is instructed to preempt the thread block in question, the context is swapped to memory (see next subsection for details), and meta-data for the thread block is pushed onto a *victim queue* maintained by the TBS, if it is not full. If the victim queue is full, no thread blocks are preempted until space is made available.

If, upon update of the RUT, the estimated resource usage has decreased, the thread block searches through the victim queue in last-in-first-out order for a thread block that is not expected to oversubscribe the SM and context-swaps the thread block into the specified SM. If no thread blocks in the victim queue meet the criteria, the TBS checks to see if the next thread block which has not yet started execution will oversubscribe the SM, and issues if appropriate. The victim queue is searched first and in a LIFO order to take advantage of locality in the event that the thread block is issued to an SM which shares part of the cache hierarchy of the previous SM.

A. Thread Block Preemption

Preempting thread blocks on a GPU has been demonstrated as an effective way to ensure quality of service when running multiple kernels [20] [21] [22]. We make use of context switching, which involves moving the contents of the register file, scratchpad, and other data encapsulating the state of running thread blocks between memory and SMs. We briefly summarize alternative methods for thread block preemption in Appendix A.

Table I includes the average total context size for a thread block in each application, which is calculated as the amount of shared memory per block plus the thread block size multiplied by the number of registers per thread, multiplied by the register size (32 bits). The table also includes the average latency added per context switch, which was measured in our simulation infrastructure (see section V for details). As is apparent, the latency associated with preempting thread blocks in each benchmark is still an order of magnitude lower than the typical phase lengths we observe and can therefore be a reasonable cost to pay if thread blocks are better co-located and resource interference can be sufficiently reduced.

V. Methodology

We evaluate our design using GPGPU-Sim version 3.2.2 [23] whose configuration is described in table II, meant to emulate the Tesla V100 we used in previously described experiments. We modified the simulator to enable multi-kernel execution and preemption with context switching, and co-ran pairs of applications from the LonestarGPU benchmark suite Version 3.0 [5]. Because the LonestarGPU benchmarks are predominately memory bound workloads, we also co-ran the benchmarks with samples from the Parboil benchmark suite [24] which includes a more diverse set of memory and compute bound workloads. For all pairs of workloads we measure the change in overall throughput (measured in instructions committed per cycle) of the LonestarGPU kernels. We compare our solution against two variants:

System	Configuration
SMs	80 SMs, 5120 Cores, 1.6GHz Max of 2K threads per SM Max of 1K threads per TB Max 64K registers per block Max 48 KB shared memory per block 128 KB L1
Memory Subsystem	6 MB L2 900 GB/s bandwidth
Thread Scheduling	GTO

TABLE II
GPGPU-SIM SIMULATION PARAMETERS USED FOR EVALUATION

- A scheduler which has oracle knowledge of the average resource usage of the kernel as a whole for use in scheduling thread blocks, but does not track the dynamic behavior of the kernel and does not employ preemption. This provides a fair comparison to works such as Xu et al [25] which estimate kernels performance characteristics as a function of their input size.
- A scheduler which tracks the dynamic behavior of kernels as a whole and preempts when oversubscription is detected, but does not track individual thread blocks. This provides a fair comparison to works such as Park et al [17].

For input, we used datasets synthesized using the Graph500 R-MAT generator [26] using its default parameters of (A=0.57, B=C=0.19) which were used in recent analytic works on power-law graphs [27] [18]. The number of edges is set to 100,000 and the number of vertices is swept between 5,000 and 80,000.

As a heuristic, we have the SMs query the performance counters once every (context_size_in_KB / 1.1) μs, which roughly equates to five times the expected preemption latency. This information can be communicated via the TBS upon thread block dispatch. We set the size of each RUT entry to be 3 bits per metric of interest (rate of memory throttle stalls and FPU utilization), and set the total size of the table to be 16KB or about 21K entries, more than enough to ensure virtually no conflicts between different kernels. Any deviation detected with this metric over that time period will be communicated back to the TBS. We set the victim queue to be large enough to hold 1K entries.

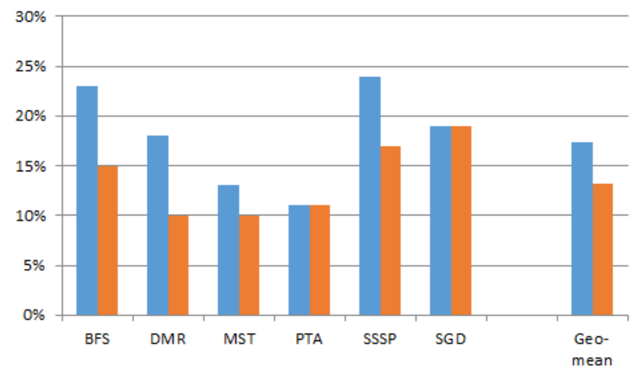

Fig. 7. Throughput increases over oracle static classification (blue) and kernel granularity profiling (orange)

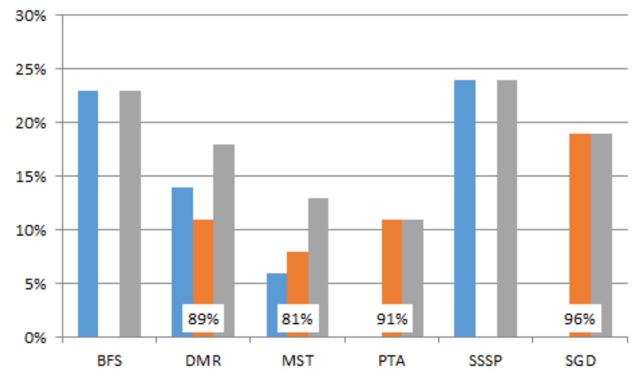

Fig. 8. Throughput increases over static classification when employing only preemption (blue), only predictive scheduling (orange), and both (grey). Annotated with correlation of subgragh densities across subsequent kernel invocations

VI. Evaluation

Figure 7 shows the throughput increase of our design over both the oracle static classification scheme and the kernel-wide monitoring solution described in section V. Each result is the average increase over all pairings and input datasets involving that benchmark. Our design, across all benchmarks, achieves an average increase in throughput of 17.1% compared to the static scheduler and 12.9% compared to the kernel-wide profiler. The magnitude of the throughput increase correlates with the range of metric deviations reported in figure 4, as benchmarks with higher variance pose the greatest hazard for interference as the data patterns change. In most cases, we achieve a better improvement over the static scheduler than over the profiler. This is because the profiler, although it does not track individual thread block metrics, can still detect coarse-grained average changes in the data set across the whole kernel and can preempt the thread blocks in cases where the variances do not cancel each other out. The kernel profiler does not perform any better than the static scheduler for either PTA or SGD, as these benchmarks consist of kernels too short for thread block preemption to be effective. To better understand the efficacy of our design, we isolated several features.

A. Effectiveness of Predictive Scheduling versus Context Switching

Figure 8 shows our design's performance when employing only the preemption or predictive scheduling strategies in isolation, and then using both together. BFS and SSSP achieve all of their throughput increases through preemption, as they only invoke their primary kernel once and so can't use the dynamic metrics to schedule future thread blocks more efficiently. Although the kernel profiling strategy is able to respond to global changes across the entire kernel, it cannot respond to deviations when individual thread blocks enter or exit dense regions of data. At the other extreme, PTA and SGD's kernels are too short to use preemption effectively and achieve all of their performance gains through scheduling enhancements. DMR and MST receive moderate speedups due to preemption, but are limited by their relatively short execution latencies and are less able to amortize the overhead of context switches.

To illustrate the effectiveness of predictive scheduling, figure 8 also annotates the workloads using predictive scheduling with the correlation of the average density of nodes processed by two TBs with the same block ID across subsequent kernel invocations. Predictive scheduling relies on the assumption that thread blocks with the same block ID will enter similar regions of the graph, i.e. we expect these correlations to be high. BFS and SSSP of course have undefined correlations since they do not have multiple kernel invocations. SGD has a very high correlation of 96%, as it is a topology driven benchmark which is guaranteed to have the same nodes processed by the same threads each iteration. Small deviations occur as the graph is morphed over time, but slowly relative to the large number of pipelined, low-latency kernel invocations. DMR and PTA are also topology driven benchmarks, but morph the underlying data structures more rapidly relative to the frequency of kernel invocations, and as such see a smaller but still substantive correlations at 89% and 91%, respectively. MST sees the lowest correlation at 81% as it is a data-driven algorithm and thus sees a greater variance in which thread blocks process which nodes. However, the use of distributed worklists keeps the correlation relatively high.

Most benchmarks receive the vast majority of their speedup from one strategy or the other and don't receive much added benefit from using both. MST is the exception, as the two strategies are often able to "catch" opportunities to better schedule threads missed by the other. On occasions where a thread is not placed on an optimal SM due to variance from worklist distribution or otherwise, preemption can still relocate the thread block, albeit with higher overhead than the other benchmarks.

B. Memory Tracking versus Compute Tracking

Figure 9 shows the performance increase when only tracking memory stalls, FPU utilization, or both together. For most benchmarks, tracking memory stalls is much more profitable than tracking FPU utilization, which is expected as these are

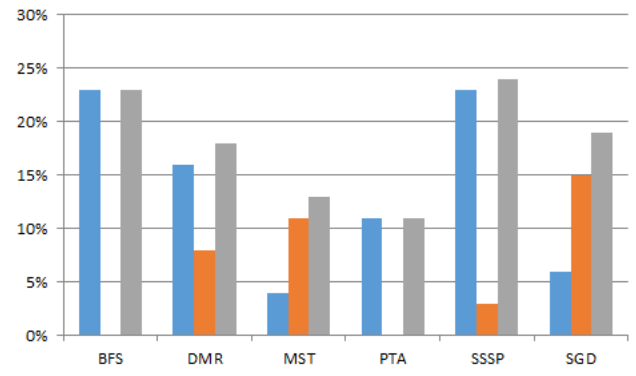

Fig. 9. Throughput increases over static classification when only tracking memory throttle stalls (blue), only tracking FPU utilization (orange), and tracking both (grey)

memory bound workloads. In particular, BFS and PTA have no floating point code. SGD and MST contain more complicated operators which make variable use of floating point operations depending on the input data structure and see more benefit from tracking FPU utilization. Although DMR and SSSP make use of floating point operations and the intensity does vary based on input structure, this variance is a largely a side effect of changing memory utilization: fewer memory stalls increases bandwidth, keeping the other functional units more occupied as a result and vice-versa. Accordingly, these see less benefit to tracking both metrics compared to other workloads, as tracking one implicitly gives information on the other.

C. Overheads

Excluding benchmarks which do not make use of preemption, we see an average throughput overhead of 2.7%. This is smaller than the overheads observed by Park et al [22] on the order of 10% because:

1) The amount of register and scratchpad space allocated by these benchmarks is smaller than the benchmarks explored in their work
2) Preemptions in their work occur on the order of $10-100\times$ more frequently than in this work due to typical phase lengths

By choosing appropriate parameters for our design, we have ensured significant performance improvements with small overhead. The increased memory traffic associated with updating the RUT and context switching across the whole GPU is 13 KB/s and 1 GB/s, respectively, which accounts for less than 1% overhead on the total available throughput of the system. The RUT and victim queue together account for 20 KB of extra SRAM shared across all SMs, which adds under 0.5% the overhead of the L2 cache.

VII. Related Work

Thread block preemption has been recognized as a powerful solution towards better resource management. Tanasic et al [20] suggest hardware extensions to make preemption on GPUs more efficient. Park et al [22] demonstrate how combining several preemption techniques and dynamically choosing between them can reduce latency and improve throughput in multiprogrammed workloads.

Many works have explored the problem of resource contention in GPUs and how to prevent it. Kayiran et al [28] explored how changing the number of thread blocks issued by compute versus memory intensive workloads reduces resource contention. Pai et al [29] and Zhong et al [16] investigate alternative static and dynamic methods of modifying kernel's execution properties to enable better utilization.

This work followed the lead of many others in exploring more sophisticated schedulers. Li et al [30] propose a co-design between the thread scheduler and cache allocation scheme to avoid cache contention without underutilizing other resources. Sethia et al [31] propose augmentations to the scheduler to prioritize memory requests from irregular, memory intensive workloads to reduce stalls. Zhao et al [32], Xu et al [25] and Jiao et al [6] investigate classification of kernels to enable better thread block placement. Park et al [17] develop a strategy to dynamically monitor a kernel's execution and preempt thread blocks when appropriate. Wu et al [33] and Chen et al [34] investigate solutions at the program and runtime level to enable better thread scheduling and preemption. To our knowledge we are the first to analyze how thread scheduling is impacted by the issues of irregular algorithms, and the first to develop a solution of tracking resource usage at the thread block level.

VIII. Conclusion

Processing diverse data patterns in irregular workloads results in significant resource usage variance across different threads. However, gradual changes prevalent in practical data sets make it possible to respond dynamically to these changes and modify scheduling decisions. By adding a hardware table to track metrics for each thread block as well as a queue to hold preempted thread blocks, we can effectively reorganize threads in response to changing data patterns. By increasing SRAM storage requirements by less than 0.5%, we can gain 17% and 13% improvements of throughput over previously proposed static and dynamic scheduling strategies, respectively.

References

[1] Jones, S., "Introduction to dynamic parallelism," *GPU Technology Conference Presentation S*, Vol. 338, 2012, p. 2012.

[2] "Unified Memory for CUDA Beginners," https://devblogs.nvidia.com/unified-memory-cuda-beginners/, Accessed: 2018-03-15.

[3] AMD, A., "Accelerated parallel processing: OpenCL programming guide," *URL http://developer. amd. com/sdks/AMDAPPSDK/documentation*, 2011.

[4] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al., "In-datacenter performance analysis of a tensor processing unit," *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ACM, 2017, pp. 1–12.

[5] Burtscher, M., Nasre, R., and Pingali, K., "A quantitative study of irregular programs on GPUs," *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, IEEE, 2012, pp. 141–151.

[6] Jiao, Q., Lu, M., Huynh, H. P., and Mitra, T., "Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS," *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, IEEE, 2015, pp. 1–11.

[7] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ACM, 2008, pp. 73–82.

[8] Fatahalian, K., Sugerman, J., and Hanrahan, P., "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, 2004, pp. 133–137.

[9] Harish, P. and Narayanan, P., "Accelerating large graph algorithms on the GPU using CUDA," *International conference on high-performance computing*, Springer, 2007, pp. 197–208.

[10] Méndez-Lojo, M., Nguyen, D., Prountzos, D., Sui, X., Hassaan, M. A., Kulkarni, M., Burtscher, M., and Pingali, K., "Structure-driven optimizations for amorphous data-parallel programs," *ACM Sigplan Notices*, Vol. 45, ACM, 2010, pp. 3–14.

[11] Kim, J. and Batten, C., "Accelerating irregular algorithms on gpgpus using fine-grain hardware worklists," *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2014, pp. 75–87.

[12] Nasre, R., Burtscher, M., and Pingali, K., "Data-driven versus topology-driven irregular computations on gpus," *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, IEEE, 2013, pp. 463–474.

[13] Luo, L., Wong, M., and Hwu, W.-m., "An effective GPU implementation of breadth-first search," *Proceedings of the 47th design automation conference*, ACM, 2010, pp. 52–55.

[14] Nasre, R., Burtscher, M., and Pingali, K., "Morph algorithms on GPUs," *ACM SIGPLAN Notices*, Vol. 48, ACM, 2013, pp. 147–156.

[15] Mendez-Lojo, M., Burtscher, M., and Pingali, K., "A GPU implementation of inclusion-based points-to analysis," *ACM SIGPLAN Notices*, Vol. 47, No. 8, 2012, pp. 107–116.

[16] Zhong, J. and He, B., "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 25, No. 6, 2014, pp. 1522–1532.

[17] Park, J. J. K., Park, Y., and Mahlke, S., "Dynamic resource management for efficient utilization of multitasking GPUs," *ACM SIGOPS Operating Systems Review*, Vol. 51, No. 2, 2017, pp. 527–540.

[18] Sundaram, N., Satish, N., Patwary, M. M. A., Dulloor, S. R., Anderson, M. J., Vadlamudi, S. G., Das, D., and Dubey, P., "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, Vol. 8, No. 11, 2015, pp. 1214–1225.

[19] Satish, N., Sundaram, N., Patwary, M. M. A., Seo, J., Park, J., Hassaan, M. A., Sengupta, S., Yin, Z., and Dubey, P., "Navigating the maze of graph analytics frameworks using massive graph datasets," *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, ACM, 2014, pp. 979–990.

[20] Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., and Valero, M., "Enabling preemptive multiprogramming on GPUs," *ACM SIGARCH Computer Architecture News*, Vol. 42, IEEE Press, 2014, pp. 193–204.

[21] Menon, J., De Kruijf, M., and Sankaralingam, K., "iGPU: exception support and speculative execution on GPUs," *ACM SIGARCH Computer Architecture News*, Vol. 40, IEEE Computer Society, 2012, pp. 72–83.

[22] Park, J. J. K., Park, Y., and Mahlke, S., "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2015, pp. 593–606.

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

[24] Stratton, J. A. et al., "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign.

[25] Xu, Q., Jeon, H., Kim, K., Ro, W. W., and Annavaram, M., "Warped-slicer: efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2016, pp. 230–242.

[26] Murphy, R. C., Wheeler, K. B., Barrett, B. W., and Ang, J. A., "Introducing the graph 500," *Cray Users Group (CUG)*, Vol. 19, 2010, pp. 45–74.

[27] Satish, N., Sundaram, N., Patwary, M. M. A., Seo, J., Park, J., Hassaan, M. A., Sengupta, S., Yin, Z., and Dubey, P., "Navigating the maze of graph analytics frameworks using massive graph datasets," *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, ACM, 2014, pp. 979–990.

[28] Kayıran, O., Jog, A., Kandemir, M. T., and Das, C. R., "Neither more nor less: optimizing thread-level parallelism for GPGPUs," *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, IEEE Press, 2013, pp. 157–166.

[29] Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R., "Improving GPGPU concurrency with elastic kernels," *ACM SIGPLAN Notices*, Vol. 48, ACM, 2013, pp. 407–418.

[30] Li, D., Rhu, M., Johnson, D. R., O'Connor, M., Erez, M., Burger, D., Fussell, D. S., and Redder, S. W., "Priority-based cache allocation in throughput processors," *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2015, pp. 89–100.

[31] Sethia, A., Jamshidi, D., and Mahlke, S., "Mascar: Speeding up GPU warps by reducing memory pitstops," *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, Feb 2015, pp. 174–185.

[32] Zhao, X., Wang, Z., and Eeckhout, L., "Classification-Driven Search for Effective SM Partitioning in Multitasking GPUs," *Proceedings of the 2018 International Conference on Supercomputing*, ACM, 2018, pp. 65–75.

[33] Wu, B., Chen, G., Li, D., Shen, X., and Vetter, J., "Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations," *Proceedings of the 29th ACM on International Conference on Supercomputing*, ACM, 2015, pp. 119–130.

[34] Chen, G., Zhao, Y., Shen, X., and Zhou, H., "EffiSha: A software framework for enabling effficient preemptive scheduling of GPU," *ACM SIGPLAN Notices*, Vol. 52, ACM, 2017, pp. 3–16.

Appendix A
Thread Block Preemption Strategies

Preemption strategies usually fall into one of three classifications:

- Context switching: Due to the massive number of threads present on SMs, true context switching is generally avoided. It requires storing the register file and scratchpad to a designated location in memory, which greatly increases the latency of preemption [22].
- Draining [20] [22]: Draining allows all currently running thread blocks to complete until the SM is empty, and then swaps in a new set of thread blocks. As has already been established, waiting for thread blocks to finish is too costly for long running kernels such as BFS and SSSP to be a general solution.
- Flushing [22]: Flushing drops the currently running thread blocks and restarts them at the beginning of their execution on a new SM at a later time. This ensures low latency swapping, but requires certain properties of the kernel at the point of preemption: it must not have made any stores to memory that could affect the values of its earlier loads, and it cannot have executed atomic operations. Otherwise, the thread block may have different execution when rerun from the beginning. Because many irregular workloads are morph algorithms (e.g. DMR and MST) which alter the input data structures as they execute, they may not meet this criteria and flushing cannot be used.

Context switching is the only general solution and is thus what is used for the design in this paper.

Appendix B
Cataclysmic Thread Purging

Upon thread preemption, our strategy marks context data as non-cacheable. Figure 10 shows a possible outcome of context swapping when the state is cached in the L1 and L2 levels of the memory hierarchy. This results in a generalized form of cache thrashing, where not just the contents of memory are brought in and out of the cache in an unstable manner, but also the contexts of the threads themselves.

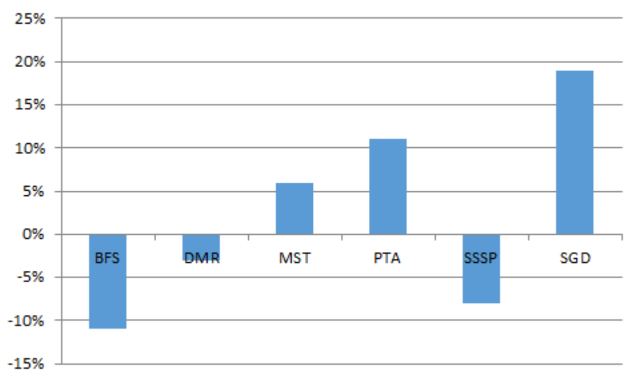

Fig. 10. Negative impact of L1 and L2 caching during context swapping

As shown in table I, many of the kernels have context sizes which are a significant portion (10% or more) of the L1 cache. When cache entries are allocated as a result of saving or restoring a thread block's context, it evicts a significant portion of the cache. In memory-intense workloads, particularly BFS and SSSP, this has the effect of significantly increasing pressure on the load-store units and increasing the probability that further thread blocks are preempted as resources are saturated. As more thread blocks are evicted, more of the cache is displaced and still more pressure is placed on the memory system, resulting in a chain reaction where a large portion of the resident thread blocks are evicted. We refer to this phenomena as *cataclysmic thread purging*. Those benchmarks that do not make as much use of preemption (e.g. short running kernels such as PTA and SGD) see less effects, but the overall throughput improvement across benchmarks is reduced to just 2%.

As a result, our design marks context saving and restoring memory operations as "non-cacheable" to prevent such an event.