2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists

Ji Kim and Christopher Batten

School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{jyk46,cbatten}@cornell.edu

Abstract—**Although GPGPUs are traditionally used to accelerate workloads with regular control and memory-access structure, recent work has shown that GPGPUs can also achieve significant speedups on more irregular algorithms. Data-driven implementations of irregular algorithms are algorithmically more efficient than topology-driven implementations, but issues with memory contention and memory-access irregularity can make the former perform worse in certain cases. In this paper, we propose a novel fine-grain hardware worklist for GPGPUs that addresses the weaknesses of data-driven implementations. We detail multiple work redistribution schemes of varying complexity that can be employed to improve load balancing. Furthermore, a virtualization mechanism supports seamless work spilling to memory. A convenient shared worklist software API is provided to simplify using our proposed mechanisms when implementing irregular algorithms. We evaluate challenging irregular algorithms from the LonestarGPU benchmark suite on a cycle-level simulator. Our findings show that data-driven implementations running on a GPGPU using the hardware worklist outperform highly optimized software-based implementations of these benchmarks running on a baseline GPGPU with speedups ranging from 1.2–2.4$\times$ and marginal area overhead.**

I. Introduction

General-purpose graphics-processing units (GPGPUs) exploit both data-level and thread-level parallelism to accelerate workloads that operate on regular data structures such as arrays and dense matrices. GPGPUs are less suited for operating on irregular data structures such as graphs and trees, due to the overhead of control and memory-access divergence which can severely limit the amount of work amortized across threads. However, recent efforts have shown that it is still possible to obtain significant speedups over multicore implementations for these irregular workloads using clever software optimizations [12, 13, 18–20, 22, 23].

A common class of irregular algorithms iteratively apply an operator to nodes in a graph which can modify the graph and generate more work for the next iteration [22]. When mapping such algorithms to GPGPUs, two common approaches are taken: topology-driven and data-driven. In topology-driven implementations, the thread index is used to determine which nodes to operate on and all nodes are visited whether or not useful work is required at the node. In data-driven implementations, only nodes at which useful work is required are visited and a shared software worklist (SWWL) is accessed to determine which nodes to operate on. Data-driven implementations are algorithmically more efficient and are better at maintaining high work efficiency (i.e., less work required to complete the kernel) but can be inferior to topology-driven implementations due to two key weaknesses. Data-driven implementations require a SWWL that needs to be accessed by thousands of threads resulting in high memory contention, and are prone to increased memory-access irregularity due to the random nature of dynamic work distribution. By utilizing both existing and in-house software optimizations, we were able to derive highly optimized versions of irregular algorithms mapped to GPGPUs. Although aggressive software optimization can help mitigate some of the weaknesses of data-driven implementations, we found that this does not completely eliminate issues with memory contention and can come at the cost of poor load balancing. As such, it is not always clear which implementation will be optimal on a GPGPU because the performance of either approach is dependent on many variables. We verify this unpredictability in our evaluation of both the topology-driven and data-driven implementations of several challenging irregular algorithms in the LonestarGPU (LSG) benchmark suite [4] on an NVIDIA Fermi-class Tesla C2075 GPGPU.

Dynamic parallelism in NVIDIA Kepler and Maxwell [24–26] is one approach to addressing load balancing concerns on GPGPUs by exposing nested parallelism at a coarse granularity. Unfortunately, coarse-grain work distribution might not be ideal when most of the work is dynamically generated as in the irregular algorithms described above. The overhead of coarse-grain recursive kernel launches could outweigh the benefits from load balancing. In this paper, we propose augmenting GPGPUs with a fine-grain hardware worklist (HWWL) to address the classic weaknesses of data-driven implementations. The HWWL is exposed to software as a shared worklist, but is implemented as distributed hardware queues tightly coupled with the GPGPU lanes to avoid memory accesses when interacting with the worklist. A hardware work redistribution unit is used to provide fine-grain, dynamic load balancing. To support cases when the total work is greater than the capacity of the HWWL, a hardware-based virtualization mechanism is used to allow seamlessly spilling work to memory. We examine both a naive on-demand scheme as well as a more sophisticated interval-based scheme for refilling the HWWL from an overflow buffer in memory.

In order to evaluate our techniques, we first provide a limit study of the maximum potential of our techniques using an idealized model of a fine-grain HWWL. We then analyze a realistic HWWL implementation using sensitivity studies to explore the effects of work redistribution schemes and virtualization on the system. Our evaluation shows that data-driven implementations running on a GPGPU with a realistic HWWL out-perform both topology- and data-driven software-based implementations running on a baseline

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.24

IEEE computer society

```
__global__ void
topo_driven( Node* nodes, bool* done_ptr ) {
  int tidx = blockIdx.x * blockDim.x + threadIdx.x;
  Node my_node = nodes[tidx];
  if ( check( my_node ) ) {
    compute( my_node )
    *done_ptr = false;
  }
}

int main() {
  bool done = false;
  while ( !done ) {
    done = true;
    topo_driven<<<N>>>( nodes, &done );
  }
}
```

(a) Topology-Driven Example Kernel

```
__global__ void
data_driven( Node* nodes, WL* wl ) {
  while ( widx = wl->pull() ) {
    Node my_node = nodes[widx];
    compute( my_node );
    for ( i = 0; i < my_node.num_neighbors; i++ ) {
      int neighbor_idx = my_node.neighbor_idx( i );
      if ( check( nodes[neighbor_idx] ) )
        wl->push( neighbor_idx );
    }
  }
}

int main() {
  init_wl<<<N>>>( nodes, wl );
  data_driven<<<M>>>( nodes, wl );
}
```

(b) Single-Buffered Data-Driven Example Kernel

Figure 1. Topology- vs. Data-driven Implementations of Example Kernel – The check operator determines if nodes are active and the compute operator performs work at these nodes. This can activate previously inactive nodes in the next super-step. Execution completes when all nodes are inactive. `N` = number of nodes, `M` = max number of HW threads, `WL` = SWWL class. In the topology-driven example, there is an acceptable race condition when updating `done_ptr`. In the data-driven example, `M` threads are spawned with the kernel since every thread will stay in the loop until no more work is in the worklist. An initialization kernel populates the worklist with active nodes before the main kernel is called. In this example, we assume that a pull returns zero when the worklist is empty.

GPGPU, with speedups ranging from 1.2–2.4×. Further studies show that a HWWL scales favorably with the number of cores and only introduces a marginal area overhead.

Section II discusses the tradeoffs between implementing irregular algorithms using a topology-driven or data-driven approach on GPGPUs. In Section III, we describe the details of our techniques, including a fine-grain HWWL baseline, work redistribution, and virtualization. Section IV evaluates the performance of our techniques compared to traditional software-based implementations of irregular algorithms mapped to GPGPUs. Section V provides first-order analyses on scalability and area overheads. Finally, Section VI discusses related work. The primary contributions of this study are: (1) we explore both existing and in-house optimizations to derive highly optimized software-based implementations of challenging irregular algorithms mapped to GPGPUs; (2) we propose a novel fine-grain hardware worklist for GPGPUs that addresses the classic weaknesses of data-driven implementations using a SWWL; (3) we propose several work redistribution schemes for dynamic load balancing; (4) we propose two virtualization schemes for spilling work to memory; and (5) we use a detailed cycle-level microarchitectural simulator to explore the design space and demonstrate the promise of our techniques for accelerating challenging irregular algorithms on GPGPUs.

II. Mapping Irregular Algorithms to GPGPUs

In this section, we begin by analyzing a simple example kernel that captures some of the key characteristics of irregular algorithms. The goal is to develop insight into the tradeoffs associated with topology- and data-driven implementations. We then verify whether this insight holds for realistic irregular algorithms and analyze how various data-driven optimizations interact with these tradeoffs. From this analysis, we derive highly optimized topology- and data-driven implementations of the benchmarks in the LonestarGPU suite [4] to use as baselines for comparing our hardware techniques.

A. Topology- vs. Data-Driven Implementations

The topology- and data-driven implementations of an example kernel representative of irregular algorithms are shown in Figure 1. The algorithm iteratively applies a compute operator (i.e., `compute()`) on a subset of nodes at which useful work is required, in an undirected, weighted graph. We refer to such nodes as *active nodes*, whereas nodes at which no useful work will be done are referred to as *inactive nodes*. We call the node ID of an active node a *work ID*. A check operator (i.e., `check()`) determines whether a node is active or inactive. The compute operator often accesses neighboring nodes and can activate inactive nodes to be processed later. Execution completes when all nodes are inactive and will not be activated again.

Figure 1(a) shows the topology-driven implementation of the example kernel. During each super-step, every thread checks to see if the node corresponding to its thread index is active. The compute operator is applied to active nodes and threads with inactive nodes exit without doing any useful work. Every kernel call represents a super-step and the kernel is invoked as long as there are active nodes in the next super-step. All nodes, both active and inactive, are visited every super-step and the number of threads is the number of nodes.

Figure 1(b) shows the data-driven implementation of the example kernel. In this case, an initialization kernel pre-checks all the nodes and populates the worklist with only active nodes. Although checking inactive nodes in the initialization is not useful, we only pay this overhead once, instead of every super-step as in the topology-driven implementation. Every thread in the main kernel pulls a work ID from the worklist with an atomic memory operation (AMO) and applies the compute operator to the corresponding active node. Newly activated nodes are pushed onto the worklist with AMOs, so that only active nodes are ever visited. Even though the check for determining new nodes to be activated is shown separate from the compute in the example, the check is usually combined with useful work in the compute phase. Although this can be done on the topology-driven implementation as well, the difference is that threads operating on inactive nodes in topology-driven implementations must pay this overhead again at the beginning of every super-step. Notice that the super-step loop seen in the topology-driven implementation is moved inside of the kernel in the data-driven implementation so that operations at nodes across multiple super-steps are overlapped. It is sufficient to only spawn a number of threads equal to the maximum number of hardware threads on the GPGPU since every thread stays in the loop until there are no more active nodes. Special consideration must be taken to prevent threads from prematurely dropping out of the computation loop.

The data-driven implementation exposes more parallelism by overlapping operations across multiple super-steps and increases work efficiency by avoiding useless work. However, the data-driven implementation is not without its weaknesses, including: high memory contention from accessing a shared worklist, and high memory-access irregularity from dynamic load balancing.

B. Benchmark Descriptions

The example kernel above represents a rough template of realistic, challenging irregular algorithms, such as those in the LonestarGPU (LSG) benchmark suite [4]. In this section, we describe the algorithm and some associated challenges for each benchmark. All of these algorithms are irregular and several of them are classified as *morph* algorithms, meaning they modify the structure of the graph by adding or removing nodes and/or edges throughout the execution of the algorithm. More information on these algorithms and traditional GPGPU mappings can be found in [18, 20, 22, 23, 27].

Breadth-first search (BFS) calculates the minimum number of edges between a source node and all other nodes in an unweighted graph. Every node keeps track of its distance from the source. The algorithm begins with one active node, the source node, which updates its neighbors with the distance from itself. This activates the neighboring nodes and this process repeats until all nodes have been updated with the minimum distance from the source node. The compute operator in BFS is relatively simple.

Barnes-Hut N-body simulation (BH) iteratively computes the position of celestial bodies changed by the gravitational force they induce on each other. Every super-step, an octree is generated to hierarchically partition the bodies into individual leaf nodes. The octree is used to approximate groups of bodies as a single source of gravitational force. Although the compute operator is relatively simple, bodies deeper in the tree generally require more work. BH is unique in that it is the only algorithm of the suite for which all nodes are active, making it more natural to map to topology-driven implementations. However, as we will see later, a data-driven implementation can help improve load balancing.

Delaunay mesh refinement (DMR) takes a Delaunay mesh and refines triangles that violate various constraints (e.g., minimum angle, Delaunay constraint) by retriangulating the bad triangle around the center of its circumcircle. Retriangulation generates more triangles, some of which might again violate the constraints. Refinement continues until there are no more bad triangles in the mesh. DMR is a morph algorithm which adds and removes triangles from the mesh. This means that special care must be taken when multiple bad triangles attempt to modify the same nodes and edges. Global barriers are often employed to facilitate conflict detection, allowing atomic execution of retriangulation [23]. The DMR compute operator is among the more complex in LSG.

Minimum spanning tree (MST) uses Boruvka's method to compute a subset of a weighted graph that spans all nodes with the minimum cost. All nodes are initialized as their own component. The algorithm iterates through components and first determines the minimum-cost path out of the component, then groups components connected by minimum-cost paths into a bigger component. This process continues until only one component is left. The compute operator becomes increasingly complex and the level of parallelism decreases as more components are merged [23]. MST is another morph algorithm that can dynamically add and remove nodes and edges from the underlying graph structure.

Survey propagation (SP) is a heuristic SAT solver that uses Bayesian inference to compute the probability of a boolean formula being true. A SAT formula is represented as a factor graph of literals and clauses with edges weighted with the probability of being true or false. The algorithm selects a literal and updates the probability of the edges. This triggers the neighboring clauses to update their literals. This process continues until probabilities converge, at which point sufficiently fixed literals are removed from the graph. The resulting graph is iterated upon again until all literals are fixed or after a maximum number of super-steps. SP is yet another morph algorithm, but one which only removes nodes and edges from the graph. Synchronization is again required to coordinate threads attempting to update the same edges.

Single-source shortest-path (SSSP) is similar to BFS except that it calculates the minimum cost between a source node and all other nodes in a *weighted* graph. One main difference with BFS is that SSSP might require more memory bandwidth since threads need to read both nodes and edges.

C. Data-Driven Optimizations

Unfortunately, although data-driven implementations are algorithmically more efficient, severe memory contention renders them impractical compared to topology-driven implementations, and they are rarely used without aggressive software optimizations [20, 22]. We can improve the viability of data-driven implementations by utilizing distributed worklists with per-thread or per-block partitions with software work redistribution, but even this technique is not enough to make data-driven implementations competitive with their topology-driven counterparts. On the other hand, topology-driven implementations are practical even without aggressive optimizations. Because of this and since the topology-driven implementations in LSG are already more consistently optimized, we focus on improving the data-driven counterparts.

The goal of this section is to identify highly optimized topology- and data-driven implementations of the benchmarks from the LSG suite [4] suitable as baseline software-based implementations to compare with our techniques. This task becomes quite difficult due to several factors: (1) the highest performing implementations of each benchmark are scattered across multiple versions of the suite, (2) only some benchmarks have both topo- and data-driven implementations, and (3) different optimizations are applied to each benchmark. The publicly released LSG1.02 only includes the topology-driven implementations. The data-driven implementations based on the same version of the suite were provided by the suite developers. LSG2.0 contains only topology-driven implementations of BH/MST/SP and only a data-driven implementation of DMR. The data-driven optimizations utilized in the LSG benchmarks are primarily from [22], but not all optimizations detailed in that study were used. To verify if there was any room for improvement, we added in-house implementations of some of these missing optimizations and measured their impact on performance. Furthermore, we fixed several bugs in the LSG1.02 benchmarks such as an issue with not all aborted bad triangles being reprocessed in DMR and a memory allocation bug in MST.

These data-driven optimizations can be grouped into four categories based on the weakness they are trying to improve: (1) reducing memory contention on pulls, (2) reducing memory contention on pushes, (3) improving load balancing, and (4) further increasing work efficiency.

Double-buffering addresses memory contention by splitting the worklist into separate pull and push partitions. Past work suggests that this is the minimum optimization required to make data-driven implementations practical [22]. Figure 2 shows the data-driven implementation of the example kernel with the double-buffering optimization. This implementation is similar to its topology-driven counterpart in that the super-step loop is external to the kernel. Every super-step, active nodes in the pull worklist are statically distributed across the threads based on the thread index. Newly activated nodes for the next super-step are pushed onto the push worklist using AMOs. Every kernel call visits all active nodes in the pull worklist, after which the pull and push worklists are swapped. The kernel is invoked as long as there are active nodes in the pull worklist at the beginning of the super-step. The main benefit of double-buffering is the elimination of AMOs on pulls which reduces memory contention. The downside is that we sacrifice parallelism due to moving the super-step loop external to the kernel. Another consequence is the loss of dynamic load balancing because threads applying compute operators at active nodes that require less work cannot steal work from other threads due to the static work distribution.

Work chunking and **atomic-reduced updates** target reducing memory contention on pushes. Working chunking amortizes AMOs by grouping multiple pushes together into a single push once the number of pushes for a thread are known. Atomic-reduced updates is a more aggressive technique that uses a per-block prefix array of the number of pushes required by each thread in a block to compute the push index, thus reducing the number of AMOs to a single AMO per block. Note that synchronization is required between steps of the prefix sum calculation. Overhead from both the calculation and synchronization can be non-trivial. The technique to use should be based on balancing these overheads with the criticality of memory contention on pushes.

```
__global__ void
data_driven_2b( Node* nodes, WL* pullwl, WL* pushwl ) {
  int tidx  = blockIdx.x * blockDim.x + threadIdx.x;
  int start = tidx * N/M;
  int end   = start + N/M;
  for ( widx = start; widx < end; widx++ ) {
    Node my_node = nodes[widx];
    compute( my_node );
    for ( i = 0; i < my_node.num_neighbors; i++ ) {
      int neighbor_idx = my_node.neighbor_idx( i );
      if ( check( nodes[neighbor_idx] ) )
        wl->push( neighbor_idx );
} } }

int main() {
  init_wl<<<N>>>( nodes, pullwl );
  while ( pullwl->size() > 0 ) {
    data_driven_2b<<<M>>>( nodes, pullwl, pushwl );
    swap( pullwl, pushwl );
  }
}
```

Figure 2. Double-Buffered Data-Driven Example Kernel – A modified data-driven implementation of the example kernel with the double-buffering optimization to eliminate atomic memory operations on pulls. For simplicity, a naive work distribution calculation is used to determine each thread's share of work based on the thread index. Note that pushes still use AMOs. Execution of each kernel completes when the pull worklist is empty. The pointers to the pull and push worklists are swapped after each kernel call. The kernel is invoked until all nodes are inactive (i.e., both worklists are empty.)

Work donating seeks to improve load balancing, which is only relevant when using double-buffering which sacrifices dynamic load balancing. This technique allows threads to donate excess work to other threads within the same block via a special donation worklist residing in shared memory. Work is donated before any computation occurs by estimating the amount of work each thread has and pushing work onto the donation worklist if this amount is greater than the block's average. The estimation operator can vary in complexity and effectiveness depending on the benchmark. Threads that finish applying operators at their share of active nodes earlier can pull more active nodes from the donation worklist.

Variable kernel configuration aims to increase work efficiency by adjusting the number of threads spawned by each kernel call based on how many active nodes are in the worklist. This primarily helps when the number of active nodes for the super-step is less than the number of HW threads by only spawning the number of threads necessary and eliminating the overhead of threads pulling from an empty worklist.

Table I shows the optimizations applied to the versions of LSG examined in this section. As certain optimizations were not available for some benchmarks, we added in-house implementations of atomic-reduced updates and work donating to the benchmarks that were missing them in LSG1.02. The resulting modified LSG suites are called LSG+A and LSG+W, respectively.

Table I. Data-Driven Optimizations of LSG Benchmarks

	LSG1.02						**LSG2.0**			**LSG+A**						**LSG+W**			
Opts	**BFS**	**BH**	**DMR**	**MST**	**SP**	**SSSP**	**BFS**	**DMR**	**SSSP**	**BFS**	**BH**	**DMR**	**MST**	**SP**	**SSSP**	**BH**	**DMR**	**MST**	**SP**
DBF	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
WCH	X		X	X	X	X											X	X	X
ARU							X	X	X	X	X	X	X	X	X				
WDO	X					X				X					X	X	X	X	X
VKC	X		X	X	X	X				X	X	X	X	X	X	X	X	X	X

Data-driven optimizations for various versions of LSG. DBF = double-buffering, WCH = work chunking, ARU = atomic-reduced updates, WDO = work donating, VKC = variable kernel configuration. LSG1.02+A = LSG1.02 benchmarks with in-house implementation of ARU instead of WCH. LSG1.02+W = LSG1.02 benchmarks with in-house implementation of WDO.

Table II. GPU Performance and Statistics of LSG Benchmarks

	Performance						**Statistics**					
	Topo		**Data**				**Dyn Insts**		**L1C Miss**		**Reqs/Load**	
Apps	**1.02**	**2.0**	**1.02**	**2.0**	**+A**	**+W**	**topo**	**data**	**topo**	**data**	**topo**	**data**
BFS	**1.0**	0.2	2.6	**4.9**	2.3		1.7e9	8.0e7	58.6	73.6	2.0	23.4
BH	1.0	**1.2**	1.0		1.0	**1.4**	2.0e8	1.6e8	44.3	64.0	7.6	6.9
DMR	**1.0**		**2.8**	0.1	2.7	2.7	6.8e7	3.9e7	31.3	40.7	4.4	4.4
MST	1.0	**2.2**	**0.1**		0.1	0.1	1.4e9	1.5e9	23.3	65.7	1.3	14.1
SP	**1.0**	0.6	1.2		**2.2**	0.5	1.6e8	1.2e7	68.1	72.3	11.4	25.0
SSSP	**1.0**	0.2	**1.2**	0.3	1.2		7.5e9	7.4e9	63.1	77.9	1.8	19.0

Performance for different LSG versions are shown on the left. Results normalized to corresponding topology-driven LSG1.02 implementations. Best version of each implementation of each benchmark are shown in bold. Useful statistics are shown on the right. Dyn Insts = number of dynamic instructions; L1C Miss = L1 data cache miss rate (%); Reqs/Load = number of memory requests generated per load.

D. *Performance Analysis*

The performance comparison of topology-driven and data-driven implementations of the various LSG suite versions (including those with our in-house optimizations) running on an NVIDIA Fermi-class Tesla C2075 GPGPU are shown in Table II. The results are speedups normalized against the corresponding LSG1.02 topology-driven implementation of the benchmark. Note that the topology-driven implementations are also highly optimized and more consistently apply the optimizations outlined in [22]. The best topology-driven and data-driven implementations for each benchmark are bolded. The highest performing implementations of the benchmarks are concentrated in LSG1.02. Only the topology-driven BH and MST, as well as the data-driven BFS are better in LSG2.0. The only case where atomic-reduced updates significantly improves performance is in SP, where the L1 data cache miss rate is reduced by 15%. This suggests that SP is more affected by decreased locality due to a more random work distribution compared to other benchmarks. Work donating does not improve performance in DMR, MST, or SP because it is difficult to accurately estimate the work required per thread in these benchmarks. However, this is less of an issue for BH, which allows work donating to produce respectable speedups by improving load balancing across a thread block. As an aside, it is interesting to note that many of the optimized LSG benchmarks running on GPGPUs have been shown to have roughly 2–3× speedups over traditional multicore systems [23].

With aggressive software optimizations, the data-driven implementations are generally able to outperform topology-driven implementations. However, even highly optimized data-driven implementations are not optimal in every situation. MST does worse using a data-driven implementation due to duplicate entries in the worklist [22]. Table II also shows useful statistics collected using detailed performance counters for the *best* topology- and data-driven implementations of the LSG benchmarks. Much of the intuition behind the tradeoffs between topology- and data-driven implementations discussed in Section II-A are reflected in these statistics. For instance, lower dynamic instruction counts illustrate the higher work efficiency of data-driven implementations. The potential impact on memory-access irregularity can be seen in the increased memory requests per load in the data-driven implementations of BFS, MST, SP, and SSSP. Similarly, the increase in L1 data cache misses in data-driven implementations suggest that locality can suffer from random work distribution as well.

From these findings, we conclude that although topology-driven implementations beat naive data-driven implementations in many cases, data-driven can outperform topology-driven with extensive software optimizations. However, even highly optimized data-driven implementations still suffer from substantial weaknesses which can limit their efficiency. Furthermore, significant effort is required to determine and implement the ideal combination of optimizations for each data-driven implementation. In general, determining the ideal approach can be difficult and is influenced by several factors including dataset size, memory criticality, complexity of the check operator relative to the compute operator, and the software worklist implementation. Next, we propose a fine-grain HWWL to address the weaknesses of highly optimized data-driven implementations, which may simplify choosing the optimal approach.

III. Fine-Grain Hardware Worklists for Accelerating Irregular Algorithms

The goal of our proposed techniques is to remedy the weaknesses of SWWL-based data-driven implementations while maintaining the algorithmic efficiency of data-driven implementations. We target two primary weaknesses: (1) memory contention on pushes, and (2) suboptimal load balancing. A fine-grain HWWL addresses memory contention by utilizing distributed hardware worklists to reduce memory accesses when interacting with the worklist. Our hardware work redistribution mechanism addresses load balancing by dynamically distributing work to threads before they become idle. We also discuss potential mechanisms for addressing memory-access irregularity in Section VII. For work that does not fit in the HWWL, we provide a hardware-based virtualization mechanism to seamlessly spill work to memory.

A. *ISA Modifications*

The HWWL is exposed to software as a shared worklist. Table III outlines the ISA modifications added for interact-

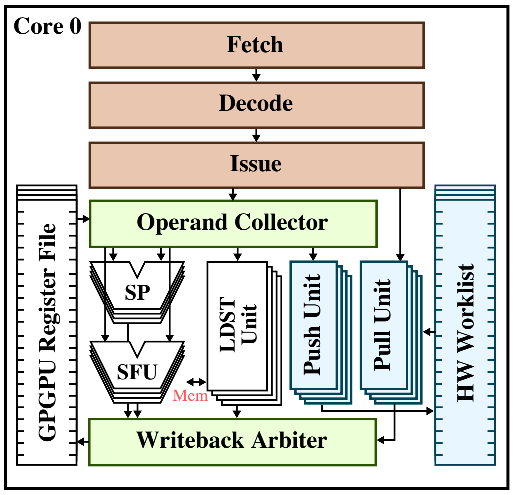


Figure 3. Modified GPGPU Pipeline Diagram – Simplified pipeline diagram of a single GPGPU core with the modifications required to support accessing the hardware worklist. Accesses to the GPGPU register file must still arbitrate for ports through the operand collector.

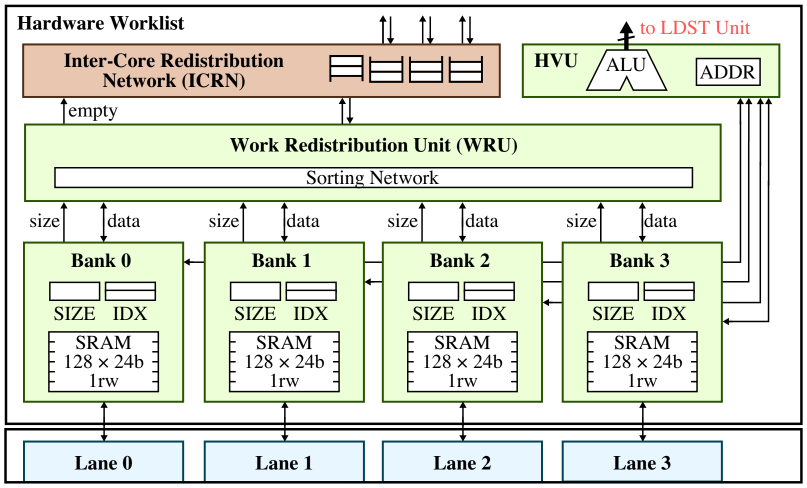


Figure 4. Hardware Worklist Microarchitecture Diagram – Interface between the GPGPU lanes and the HWWL. The lanes access the distributed HWWL banks which are managed by the work redistribution unit (WRU). The HWWL virtualization unit (HVU) handles reads/writes with the overflow buffer. The inter-core redistribution network (ICRN) is a simple tree network that facilitates work distribution between cores.

TABLE III. ISA MODIFICATIONS FOR HWWL SUPPORT

Instruction	Description
`wlinit r_d, r_s`	Initializes overflow buffer for virtualization.
`wlcfg  r_s`	Configure partition mode (0=single,1=double).
`wlpull r_d, r_s`	Pulls work ID from HWWL, if local bank is empty: return WAIT if there is more work in system, or DONE otherwise.
`wlpush r_s, r_t`	Pushes work ID to HWWL, throws exception if overflow buffer is full.

ing with a HWWL. A SWWL-based data-driven implementation of a benchmark can be adapted to use a HWWL by simply linking to a new worklist implementation that uses appropriate instructions. In addition, the *overflow buffer*, a set of per-core arrays of work IDs used during virtualization, must be set up using the `wlinit` instruction with the address and size of memory allocated by the user. This step is usually completed as part of the initialization kernel. The ISA also supports both single-buffered and double-buffered data-driven implementations which can be configured using the `wlcfg` instruction. When in double-buffer mode, separate pull and push worklists are automatically swapped after every kernel call as long as the pull worklist is empty. Pulls will return a special WAIT token if the local worklist for the thread is empty but there is more work in the system. This prevents threads from prematurely exiting the computation loop due to suboptimal load balancing and missing a chance to pull an active node in the future. Completed threads will pull a special DONE token (i.e., no more work in the system), allowing threads to exit the computation loop. When in single-buffer mode, additional logic is required to prevent threads exiting before more work is pushed into the unified pull/push worklist. For this study, we focus on analyzing the double-buffered variant. We classify the state of hardware threads based on the response of the HWWL: threads which pull a work ID are *active*, threads which pull WAIT are *waiting*, and threads which pull DONE are *complete*. Pushes will cause a hardware exception when both the HWWL and the in-memory overflow buffer are full, implying that not enough memory was allocated to hold the amount of work generated.

B. HWWL Baseline Microarchitecture

Figure 3 shows a simplified pipeline of a GPGPU core with the modifications required for interacting with a HWWL. Pulls move work IDs from the HWWL to the GPGPU register file (GRF). Since the HWWL is banked by lane such that each lane only accesses its own bank, pulls can bypass the operand collector and be directly sent to the *pull unit* to read work IDs. However, pulls must still arbitrate for a write port when writing to the GRF. Conversely, pushes move work IDs from the GRF to the HWWL. In this case, pushes must be issued to the operand collector to read work IDs from the GRF, after which they are issued to the *push unit* which writes the work IDs to the HWWL. Pushes do not need to arbitrate for a write port due to the per-lane HWWL banks.

Figure 4 shows a microarchitectural diagram of the interface between a HWWL and the GPGPU lanes in a single GPGPU core. In order to reduce memory accesses when interacting with a worklist, we implement the HWWL as distributed per-lane 1rw SRAM banks that function as FIFO queues. We only require one port since pulls and pushes cannot happen on the same cycle. We explore the effect of varying bank sizes in Section IV-E. Each HWWL entry stores a 24b work ID, supporting up to 17M unique work IDs. Although this is sufficient to run the benchmarks in our evaluation, we recognize that the entry size may have to be increased at design time to support workloads with even more unique work IDs. The HWWL can be used as a single worklist at full capacity or two worklists at half capacity. Access pointers for each partition are managed separately. In single-buffer mode, both pulls and pushes access the same worklist, whereas in double-buffer mode, pulls and pushes access their corresponding partition. Note that since the baseline design has no work redistribution, a pull from an empty bank will always return DONE since the bank will not be refilled again.

C. *HWWL Work Redistribution*

The baseline design obviously suffers from poor load balancing since a few banks can hog a majority of the work and leave other banks empty. This wastes resources since hardware threads exit without doing any useful work. Hardware work redistribution allows work to be transferred between banks to avoid this case as much as possible.

With work redistribution, we want threads that pull from an empty bank to *wait* if there is more work in other banks that might be redistributed to its empty bank later, instead of exiting. In such cases, the HWWL returns a WAIT token instead of a DONE token; DONE is only returned when there is no more work in the system and all threads are waiting. It is up to software to define what a waiting thread does, but usually this involves the thread accessing the worklist again until it receives a work ID or DONE token. Waiting threads in a warp diverge from active threads and reconverge when the active threads pull again. A per-core *empty bit* keeps track of whether or not all banks in the core are empty and is broadcasted to other cores in the system. On a pull, the HWWL returns a DONE token when all empty bits are set, otherwise it returns a WAIT token. Note that waiting threads lower work efficiency by doing useless work.

There are two policies in determining when redistribution should occur. The first is *interval-based*, where work is redistributed every sampling interval, and the second is *on-demand*, where the WRU only redistributes on a pull or push. The former has better load balancing at the cost of energy, whereas the latter is more energy-efficient but sacrifices some load balancing. Another serious drawback of on-demand redistribution is that it requires more ports on the HWWL banks since pushes and pulls need to happen simultaneously with redistribution. On the other hand, interval-based schemes can share a single port by delaying or skipping redistribution when pushes and pulls happen on the same cycle. In this study, we focus on interval-based redistribution schemes to maximize load balancing and reduce area overhead.

Hardware work redistribution is implemented with per-core work redistribution units (WRU) connected by an inter-core redistribution network (ICRN). The WRU aggregates the empty bits as well as other scheme-specific information used in determining how to redistribute work. The WRU will always try to redistribute work within its own core before resorting to inter-core redistribution to minimize the overhead of using the ICRN. The ICRN is implemented as a tree topology with a hub at the root of the tree to connect all cores in the GPGPU. As such, it takes two hops to transfer work between any two cores in the ideal case (i.e., ignoring contention). Each core has separate input and output ports to the ICRN, and thus can inject one work ID and receive one work ID per cycle.

The **threshold-based** redistribution scheme is the simplest scheme in which the *greedy* banks, which have more work than the threshold, donate work to the *needy* banks, which have less work than the threshold. In this scheme, each HWWL bank emits an additional *local greedy bit* to the WRU which is set if it has more work than the threshold. Each core also broadcasts a *global greedy bit* to other cores which is set if the core has more greedy banks than needy banks. When redistribution is triggered, the greedy banks donate work to any needy banks within the same core first in a round-robin manner. If there are more greedy banks than needy banks, one of the greedy banks sends work to another core which does not have the global greedy bit set.

The **local sorting-based** redistribution scheme increases hardware complexity to achieve better load balancing. In this scheme, each HWWL bank relays the amount of work it has to the WRU. A sorting network in the WRU uses this information to rank the banks in order of increasing amount of work. When redistribution is triggered, the banks with the most work donate work to any banks which have less work than the threshold (i.e., needy banks). This means that even needy banks are able to donate work to other needy banks which helps load balancing when all banks are needy. In such cases, the half of the banks with the most work donate work to the half of the banks with the least work. Overall, this scheme encourages more even work distribution and prevents a few banks from accumulating more and more work. Similar to before, this scheme uses global greedy bits to determine if inter-core work redistribution is required.

The redistribution schemes discussed thus far mainly use local, per-core information to determine how to redistribute work. Minimal information is relayed between cores (i.e., single bits) to make work redistribution scalable with increasing core count. The **global sorting-based** redistribution scheme uses a monolithic WRU to aggregate more detailed global information across cores. Although unrealistic due to scalability issues, we use this design to explore the impact of providing the WRU with global information on inter-core load balancing. In this scheme, a monolithic WRU ranks banks across all cores instead of just within a core and marks the half of the banks with the most work as greedy and the half of the banks with the least work as needy. When redistribution is triggered, greedy banks always donate to any needy banks within the core and any excess greedy banks send work to another core. This prevents banks which might have a lot of work in the global sense from getting more work because it has less work in the local sense. Banks below the work threshold are still prioritized even if they are considered globally greedy. We explore the effects of different redistribution schemes in Section IV-D.

D. *HWWL Virtualization*

For a practical HWWL design, it is necessary to provide support for work virtualization. The *HWWL virtualization unit* (HVU) for each core manages spilling and refilling work IDs to an in-memory *overflow buffer*. Work in the overflow buffer is considered as part of the HWWL for the purposes of determining the return value for HWWL pulls. The address and per-core size of the overflow buffer initialized by the `wlinit` instruction are used to calculate the per-core offset into the overflow buffer. This offset is stored in a special address register in the HVU. The HVU also has an integer ALU and special index registers for generating addresses for virtualization requests. The HVU tags and injects virtualization requests into the load-store unit of the GPGPU lanes to be issued just like any other memory access. As many virtualization requests as there is space in the load-store unit can be sent out, but normal memory requests are given priority. The load-store unit is modified to tag pending virtualization requests with an special bit that is used to route virtualization responses to the HWWL instead of the GPGPU lanes. Pushes to full banks within a core are aggregated by the HVU into a single coalesced virtualization store to the core's overflow buffer. Up to a warp's worth of pushes can be coalesced.

Virtualized pulls can also be handled using an **interval-based** or **on-demand** scheme. With an interval-based scheme, the HVU periodically checks to see if all banks in the core have at least one free entry. If so, it generates a virtualization load for a warp's worth of data. These free entries are marked as reserved, meaning they count as being occupied for the purposes of work redistribution or worklist accesses. As many virtualization requests can be injected as there are free entries in the load-store unit's internal queue. With an on-demand scheme, the HVU only generates a virtualization load on pulls to empty banks. Multiple virtualization loads can be issued up to a limit as long as there is space in the load-store unit. Up to a warp's worth of data can be coalesced into a single virtualization load. In this case, the HVU updates the scoreboard so that the pull looks like a pending load, forcing instructions dependent on the pull to wait to be issued until the HWWL is refilled. In order to prevent the pull from creating control divergence, we conservatively force all threads in a warp to wait even if only some of those threads pulled from an empty bank. Note that warp multithreading greatly helps with hiding this refill latency. In either case, the load-store unit routes any load responses tagged as virtualization loads directly to the HVU. Again, virtualization loads can be coalesced into a single load request since the data accessed in the overflow buffer is guaranteed to reside in contiguous memory addresses. This data is written back to the HWWL banks over cycles when there is no conflict with work redistribution or pushes/pulls. We explore the effects of the different virtualization schemes in Section IV-E.

An interesting observation is that work redistribution can interact with virtualization to non-trivially impact performance. An ineffective or non-existent work redistribution scheme can actually degrade performance by unnecessarily generating virtualizing requests when using on-demand virtualization. For instance, with a suboptimal work distribution, one bank in a core might be full even though another bank in the same core might be empty. In this case, a push would generate a virtualization store even though there is still space in the HWWL and a pull would generate a virtualization load even though there is still work in the HWWL. As such, it is important to fully utilize the HWWL storage by evenly distributing work in addition to preventing empty pulls.

TABLE IV. GPGPU-SIM SIMULATION PARAMETERS

Front End	8KB L1 instruction cache, greedy-then-oldest warp scheduling, 2 issue slots per cycle
Execution Core	700 MHz, 48 HW warps, 32768 registers, 16 lanes, 2 SP and 1 SFU per lane
Memory System	4-way set-associative 8KB L1 data cache, 8-way set-associative 786KB L2 data cache, 100 cycle DRAM latency

IV. EVALUATION

In this section, we describe our evaluation methodology, then explore the design space for work redistribution and virtualization to help select reasonable parameters for a HWWL. The performance of the final design and the impact of the HWWL mechanisms are compared against highly optimized software-based topology- and data-driven implementations of irregular algorithms from LSG.

A. Methodology

We model a fine-grain HWWL on a modified version of GPGPU-Sim 3.0 with a PTX front-end and realistic on-chip and off-chip memory system with four cores each with 16 lanes. We chose a smaller number of cores to enable better insight into the interaction between the HWWL and GPGPU pipeline, as well as achieve reasonable simulation times with realistic datasets. We believe the benefits of our techniques extend to larger GPGPUs as discussed in Section V-A. GPGPU-Sim is a cycle-level microarchitectural simulator with a functional/timing split to enable rapid design space exploration [2]. The configuration settings used in GPGPU-Sim are outlined in Table IV. We use highly optimized benchmarks from the LSG suite with in-house optimizations based on our analysis in Section II-D. All results in this section are normalized to the better of the software-based topology- (*topo*) and data-driven (*data*) implementations of each benchmark selected in Table II. The HWWL-based data-driven implementations are based on their SWWL counterpart and configured to run in double-buffer mode, but no other software optimizations are used.

We developed a simple software API for a shared worklist suitable for use in the data-driven implementations of our algorithms. There are two implementations of this API: (1) a pure-software implementation that serves as our SWWL baseline design, and (2) a software/hardware implementation that uses inline assembly to interact with a HWWL. By using a single software API, we are able to rapidly develop data-driven implementations on a real GPGPU platform, then easily port these implementations to our cycle-level simulator.

B. HWWL Limit Study

We study the maximum potential of our techniques by using an idealized HWWL design that uses a single infinite-capacity worklist with unlimited bandwidth to model near-perfect load balancing. For this section and the following

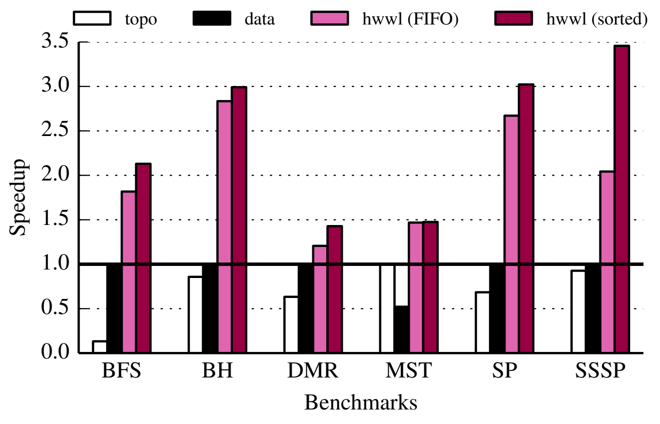


Figure 5. Comparison of Idealized HWWL Designs – Performance of a FIFO-based and sorting-based ideal HWWL design compared to highly optimized software-based implementations of LSG benchmarks to show the potential of our techniqes. Results are normalized against the better of *topo* and *data*.

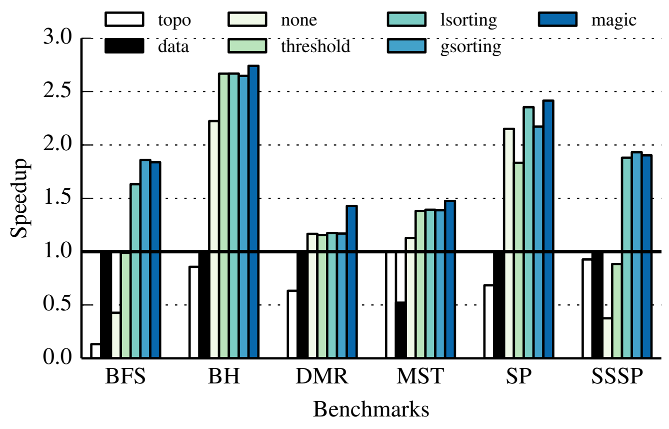


Figure 6. Comparison of Work Redistribution Schemes – Performance of an infinite-capacity HWWL using various work redistribution schemes. *none*: no work redistribution; *threshold*: local threshold-based; *lsorting*: local ranking-based; *gsorting*: global ranking-based; *magic*: idealized redistribution.

sensitivity studies, we focus on identifying general trends to guide our choices for architectural parameters of a HWWL.

Figure 5 shows the performance of a idealized HWWL design compared to *topo* and *data*. Although the absolute simulator performance between *topo* and *data* significantly differ from the GPGPU performance as expected, we can confirm that the relative trends between the two baselines are similar to what we observed in Figure II. We explore two HWWL configurations: a simple FIFO-based scheme and a magic sorting-based scheme. The latter sorts the worklist by work ID before every pull to reduce the impact of memory-access irregularity usually associated with data-driven implementations. Our findings show that the FIFO-based scheme is able to achieve speedups over the best software-based implementation ranging from 1.2–2.8×. This performance gap is attributed to improved load balancing and reduced memory requests when accessing the worklist. The sorting-based scheme further increases speedups in most cases, suggesting that memory-access irregularity contributes a respectable amount to the performance bottleneck. We focus on the FIFO-based scheme in this paper, and leave more realistic sorting mechanisms to address memory-access irregularity to future work. Overall, there is promising potential for a fine-grain HWWL, but it remains to be seen how much of this potential a realistic HWWL implementation can achieve.

C. HWWL Baseline Analysis

In this section, we evaluate the impact of using a more realistic HWWL with no work redistribution. The HWWL design here models the microarchitecture described in Section III, but each HWWL bank still has infinite capacity.

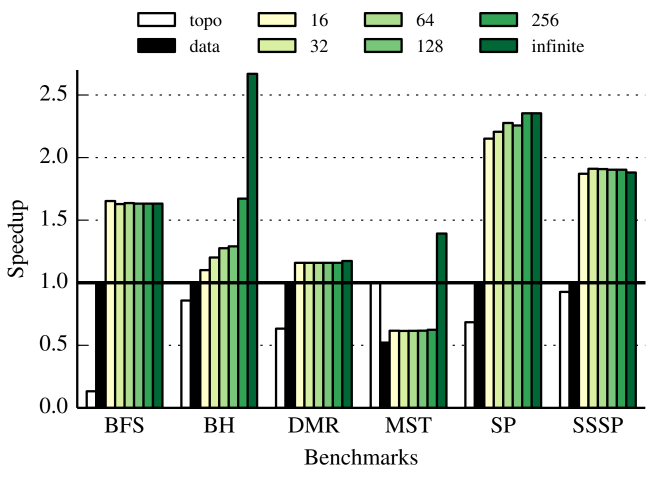
Figure 6 shows the performance of using a HWWL with no work redistribution (i.e., *none*) normalized to the best software-based implementation. Using a HWWL without any work redistribution can still achieve substantial speedups for benchmarks with inherently even load balancing like BH, DMR, and SP. The load balancing in MST gets worse in later super-steps, but with an infinite capacity, it is sufficient to reap the benefits of a HWWL without any work redistribution. The speedups can be attributed to the elimination of memory requests when accessing the worklist. Up to a 67% reduction in memory stalls and a 16% reduction in dynamic instruction count can be achieved by using a HWWL compared to a SWWL in these benchmarks, reflecting the decreased memory accesses when interacting with the worklist and the higher work efficiency from eliminating the SWWL overhead, respectively. However, benchmarks with inherently bad load balancing like BFS and SSSP get even worse load balancing without any work redistribution, resulting in lower work efficiency due to waiting threads unnecessarily spinning to obtain more work. For example, dynamic instruction counts increase by an order of magnitude for BFS and SSSP.

D. HWWL Work Redistribution Analysis

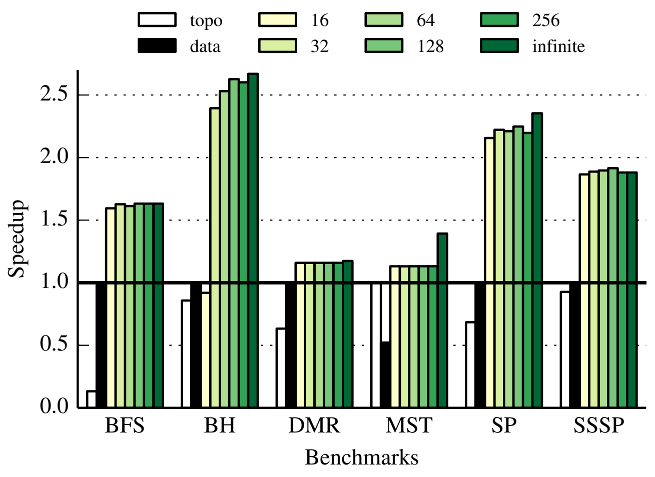
We evaluate the impact of various work redistribution schemes on a more realistic HWWL by using infinite-capacity banks to isolate the effects of work redistribution. The goal is to determine a reasonable work redistribution scheme that balances complexity and performance.

Figure 6 shows the performance of various work redistribution schemes normalized to the best software-based implementation. *threshold* and *lsorting* represent the local (i.e., only using intra-core information) threshold-based and sorting-based redistribution schemes. Both of these schemes use a work threshold of five, which our experiments suggested was a reasonable operating point. *gsorting* is the global-variant of the sorting-based redistribution scheme with a monolithic WRU that uses inter-core information. We also evaluate an idealized redistribution scheme (*magic*) that evenly distributes work across cores ignoring bandwidth constraints. Based on a preliminary study of various redistribution intervals, we selected an optimal interval of ten cycles.

Enabling any type of work redistribution generally helps most on benchmarks with inherently poor load balancing. Using the number of pulls returning a WAIT token as a proxy for the level of load balancing, BFS and SSSP displayed orders of magnitude fewer pulls returning a WAIT token by utiliz-



(a) On-Demand Virtualization



(b) Interval-Based Virtualization

Figure 7. Comparison of HWWL Bank Capacity and Virtualization Schemes – Performance of HWWL using *lsorting* work redistribution with various HWWL bank sizes using on-demand and interval-based virtualization. The corresponding infinite-capacity HWWL results are provided as a reference point. Results are normalized against the better of the *topo* and *data* implementations.

ing *lsorting* work redistribution. However, even benchmarks with inherently better load balancing can still benefit from the dynamic load balancing from work redistribution as well as improved work efficiency. Work efficiency is increased by eliminating the useless work done by waiting threads as evident by reductions of over 70% in dynamic instruction counts.

In general, *threshold* does not perform as well as the other redistribution schemes. Because this scheme allows any bank with more work than the threshold to donate work, there are often cases where a bank with work only slightly above threshold will donate work to a needy bank even though there is another bank with relatively more work. This can lead to a situation where only a few banks have a majority of the work but they are not able to donate work to all of the needy banks because of bandwidth constraints. Again, this impact is much more noticeable in benchmarks with inherently bad load balancing such as SP, where performance actually suffers due to this unfavorable work distribution.

Both of the sorting-based redistribution schemes achieve greater speedups compared to *threshold* by prioritizing the greediest bank for work donation. There are only slight performance gaps between the local and global variants for a couple benchmarks which implies that using inter-core information for redistribution usually does not improve load balancing enough to merit a monolithic WRU design. Using inter-core information might not always be beneficial as seen in SP, since local load balancing is sacrificed for global load balancing.

Overall, both *lsorting* and *gsorting* get closest to the performance of *magic*, but the performance of *gsorting* does not justify its complexity. As such, we use *lsorting* as the work redistribution scheme for the rest of the evaluation.

E. HWWL Virtualization Analysis

Now that we have a realistic work redistribution scheme, we examine the impact of HWWL virtualization on varying HWWL bank sizes to determine an optimal bank size.

Figure 7(a) shows the performance of a realistic HWWL design using *lsorting* work redistribution for a range of HWWL bank sizes with on-demand virtualization. In most cases, the performance of on-demand virtualization approaches the performance of the infinite-capacity configuration even when all work does not fit in the HWWL. This is mainly because multithreading effectively hides the refill latency as long as the compute operator is sufficiently complex. When the compute operator is relatively simple like in BH, it becomes more difficult to hide the refill latency, making the performance impact of virtualization and finite-capacity banks more apparent. MST is similar in that the compute operator starts simple and becomes more involved every super-step. Coincidentally, the most parallelism is available at the beginning and decreases every super-step, meaning the compute operator applied to most of the work is relatively simple.

Figure 7(b) shows the performance of a realistic HWWL design using *lsorting* work redistribution for a range of HWWL bank sizes with interval-based virtualization. Although in most cases interval-based virtualization is unnecessary, it is essential for benchmarks that cannot hide the refill latency with only multithreading. As we see with BH and MST, interval-based virtualization allows performance to get much closer to the infinite-capacity configuration by preemptively refilling the HWWL. This means that smaller capacities can be used to achieve the same or better speedups compared to on-demand virtualization. The factor of five reduction in pulls returning a WAIT token by using interval-based virtualization instead of on-demand virtualization for BH and MST validates how limited these benchmarks are by waiting on virtualization requests. BFS does not benefit from interval-based virtualization even though the compute operator is relatively simple because the work IDs fit within the HWWL as long as work redistribution is enabled.

In general, we see that 32 entries per bank achieves most of the speedup possible with an infinite capacity, *lsorting* work redistribution offers the best balance between performance and complexity, and interval-based virtualization hides virtualization latencies for simple compute operators. Using a realistic HWWL design with this configuration, we were able to achieve speedups ranging from 1.2–2.4× over the best software-based implementation.

V. Discussion

In this section, we provide first-order analyses of the scalability and area overheads of a fine-grain HWWL, and we revisit the performance of single-buffered data-driven implementations using a HWWL.

A. *Scalability Analysis*

We ran experiments using BFS with a very large (>1M nodes) dataset running on 4, 8, and 16 cores to explore the scalability of a realistic HWWL design. Each experiment took over one day of simulation time. We found that speedups actually increased with the core count. With 16 cores, a fine-grain HWWL improved performance by 2.5× compared to the best software-based implementation. Similar to the effect of adding more total cache capacity to achieve super-linear speedup on a CMP system, adding more cores also increases the total HWWL capacity to further accelerate performance. These results also suggest that the HWWL is largely latency insensitive as the ICRN latency tends to increase with more cores injecting work per cycle.

B. *Area Analysis of HWWL*

In this section, we provide a first-order analysis of the area overhead of using our techniques. The primary overhead of a HWWL is the additional area required by the HWWL banks. Although energy is also a concern, it can be more readily addressed by increasing the work redistribution sampling period or energy-aware redistribution schemes to reduce unnecessary data movement. Each bank can be modeled as a 1rw SRAM bank with a couple extra registers for tracking queue indices and some logic for calculating the worklist size. With 16 lanes per core, each with a 1rw SRAM with 32 entries of 24 bits, the additional state required per core is approximately 1.6 KB. In comparison, the register file of each NVIDIA Maxwell-class GPGPU core, also implemented as 1rw SRAMs, has 16,384 entries of 32 bits each for a total capacity of 64 KB [26]. Given this, the area overhead of the HWWL banks is 2.5% of the register file per core. The sorting network in the WRU can be modeled as a binary sorting network with 16 nodes and 4 stages per core which we estimate adds an overhead of 162 μm^2 based on first-order analysis using a 40 nm TSMC standard cell library.

C. *Revisiting Single- vs. Double-Buffered Approaches*

Although in Section II we deemed single-buffered data-driven implementations as unviable due to memory contention, we experimented with how a single-buffered approach would compare given that a HWWL eliminates much of the memory contention. Figure 8 shows the percent of active hardware threads over time for a single- and double-buffered data-driven implementation of BFS. The single-buffered approach is able to fully utilize all hardware threads for a majority of the execution compared to the double-buffered approach, which has a max utilization of 80%. Unfortunately, with a realistic memory system, the single-buffered approach has worse performance due to the decrease in spatial locality caused by overlapping super-steps. By using a magic memory system to eliminate this impact, we can see that the single-buffered approach does out-perform the double-buffered approach by exposing more parallelism, opening interesting avenues for future work.

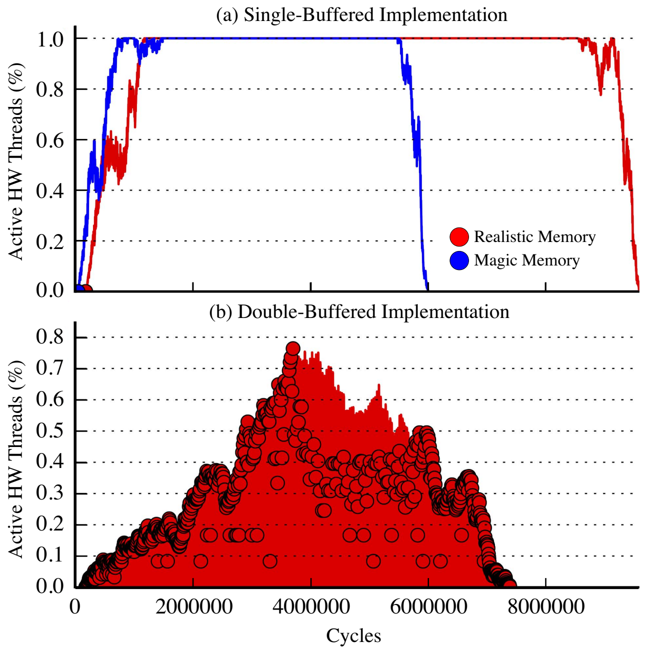


Figure 8. Single- vs. Double-Buffer Activity of BFS – The percent of active HW threads over time for single-buffered (a) and double-buffered (b) data-driven implementations of BFS. Results for both a realistic and magic memory system are only shown for the single-buffered variant as there is negligible speedup with the double-buffered variant.

VI. Related Work

To our knowledge, this work is the first to apply fine-grain hardware worklists to accelerate irregular algorithms on GPGPUs. However, there exist several studies that evaluate hardware worklists on more traditional CMP systems. Examples include work on fine-grain load balancing using specialized hardware task schedulers [16] and a hybrid hardware-software alternative to this approach allowing flexibility in choosing the scheduling policy [29]. Complementary coarse-grained approaches to work distribution [1, 13, 24–26] tend to tradeoff software complexity to ensure optimal warp construction (i.e., control/memory-access regularity). In cases where most of the work is dynamically generated on a single chip, it might be preferable to simplify work scheduling and redistribute work at a finer granularity.

There have been many comparative studies classifying irregular algorithms [15, 27] and mapping them to topology- and data-driven implementations [22], as well as software optimizations for irregular algorithms [3, 4, 12, 13, 19] and morph algorithms [23]. Topology-driven implementations could also benefit from reducing control divergence by merging warp fragments [7–9, 28] or more sophisticated reconvergence policies [5,6]. The increased memory-access irregularity in data-driven implementations could be addressed with prefetching [17], clever warp scheduling [14], or using warp fragments to hide memory latencies [21, 30].

Hardware transactional memory in GPGPUs [10] could be utilized orthogonally to help memory contention. Unified local memories in GPGPUs [11] could also be used in a similar manner to the HWWL to reduce memory accesses when interacting with the worklist, albeit with much less bandwidth and no work redistribution.

VII. Conclusions

Although data-driven implementations of irregular algorithms are algorithmically more efficient than their topology-driven counterparts, the weaknesses of using a SWWL can be a significant performance barrier even with aggressive optimizations. In this paper, we proposed a fine-grain HWWL with work redistribution and virtualization that addresses these classic weaknesses while maintaining the high work efficiency of data-driven implementations. Our results showed that even with a small area overhead, a fine-grain HWWL is able to achieve scalable speedups of up to 2.4 $\times$ over the best software-based implementation by reducing the dependence on memory requests when interacting with the worklist and providing dynamic load balancing. Using a HWWL, we saw promise for the previously impractical single-buffered data-driven implementations for which the increased resource utilization was overshadowed by the decrease in spatial locality. Both higher memory-access irregularity and lower spatial locality remain challenges to overcome. The primary direction for future work is exploring a more realistic HWWL that sorts work IDs to mitigate these weaknesses. This could potentially allow the single-buffered approach to further accelerate data-driven implementations by exposing more parallelism to the hardware. Overall, this paper opens up many exciting possibilities in specialized hardware acceleration for a class of algorithms especially challenging to map to GPGPUs.

Acknowledgments

This work was supported in part by NSF CAREER Award #1149464, an NDSEG Fellowship, and donations from Intel Corporation and NVIDIA Corporation. The authors acknowledge and thank the Lonestar Benchmark Suite team for their insightful advice, particularly Martin Burtscher, Rupesh Nasre, and Sreepathi Pai for generously providing additional code and their extensive help in understanding the various implementations of LSG benchmarks.

References

[1] Heterogeneous System Architecture: A Technical Review. AMD White Paper, 2012. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf`.

[2] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2009.

[3] J. Barnat et al. Computing Strongly Connected Components in Parallel on CUDA. *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Apr 2011.

[4] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. *Int'l Symp. on Workload Characterization (IISWC)*, Oct 2012.

[5] B. W. Coon and J. E. Lindholm. System and Method for Managing Divergent Threads in a SIMD Architecture. US Patent 7353369, Apr 2008.

[6] G. Diamos et al. SIMD Re-Convergence at Thread Frontiers. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2011.

[7] W. W. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2011.

[8] W. W. Fung et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2007.

[9] W. W. Fung et al. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Trans. on Architecture and Code Optimization (TACO)*, 6(2):1–35, Jun 2009.

[10] W. W. L. Fung et al. Hardware Transactional Memory for GPU Architectures. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2011.

[11] M. Gebhart et al. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2012.

[12] P. Harish and P. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. *Int'l Conf. on High-Performance Computing (HIPC)*, Dec 2007.

[13] S. Hong et al. Accelerating CUDA Graph Algorithms at Maximum Warp. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2011.

[14] A. Jog et al. Orchestrated Scheduling and Prefetching for GPGPUs. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2013.

[15] A. Kerr, G. Diamos, and S. Yalamanchili. A Characterization and Analysis of PTX Kernels. *Int'l Symp. on Workload Characterization (IISWC)*, Oct 2009.

[16] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2007.

[17] J. Lee et al. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2010.

[18] L. Luo, M. Wong, and W. Hwu. An Effective GPU Implementation of Breadth-First Search. *Design Automation Conf. (DAC)*, Jun 2010.

[19] M. Mendez-Loj et al. Structure-Driven Optimizations for Amorphous Data-Parallel Programs. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2010.

[20] M. Mendez-Lojo, M. Burtscher, and K. Pingali. A GPU Implementation of Inclusion-Based Points-to Analysis. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2012.

[21] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2010.

[22] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus Topology-driven Irregular Computations on GPUs. *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Apr 2013.

[23] R. Nasre, M. Burtscher, and K. Pingali. Morph Algorithms on GPUs. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2013.

[24] Dynamic Parallelism in CUDA. NVIDIA White Paper, 2012. `http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf`.

[25] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. NVIDIA White Paper, 2012. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[26] NVIDIA GeForce GTX 750 Ti. NVIDIA White Paper, 2014. `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf`.

[27] K. Pingali et al. The Tao of Parallelism in Algorithms. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2011.

[28] M. Rhu and M. Erez. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2012.

[29] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2010.

[30] D. Tarjan et al. Increasing Memory Miss Tolerance for SIMD Cores. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, Aug 2009.