GSI: A GPU Stall Inspector to Characterize the Sources of Memory Stalls for Tightly Coupled GPUs

Johnathan Alsop† Matthew D. Sinclair† Rakesh Komuravelli‡ Sarita V. Adve†
† University of Illinois at Urbana-Champaign
‡ Qualcomm Technologies, Inc.
hetero@cs.illinois.edu

Abstract

In recent years the power wall has prevented the continued scaling of single core performance. This has led to the rise of dark silicon and motivated a move toward parallelism and specialization. As a result, energy-efficient high-throughput GPU cores are increasingly favored for accelerating data-parallel applications. However, the best way to efficiently communicate and synchronize across heterogeneous cores remains an important open research question. Many methods have been proposed to improve the efficiency of heterogeneous memory systems, but current methods for evaluating the performance effects of these innovations are limited in their ability to attribute differences in execution time to sources of latency in the memory system. Performance characterization of tightly coupled CPU-GPU systems is complicated by the high levels of parallelism present in GPU codes. Existing simulation tools provide only coarse-grained metrics which can obscure the underlying memory system interactions that cause performance differences. In this work we introduce GPU Stall Inspector (GSI), a method for identifying and visualizing the causes of GPU stalls with a focus on a tightly coupled CPU-GPU memory subsystem. We demonstrate the utility of our approach by evaluating the sources of stalls in several recent architectural innovations for tightly coupled, heterogeneous CPU-GPU systems.

I. Introduction

As power and energy constraints limit further scaling of single core performance, chip designers have turned to parallelism and heterogeneity to deliver further improvements in performance and efficiency. In particular, GPUs are increasingly used alongside CPUs to accelerate applications. Recently, CPU-GPU systems are becoming increasingly tightly coupled [1], [2], [3], [4]. This shift enables GPU acceleration for an ever broader class of parallel applications, but conventional GPU systems do not provide efficient support for them. Therefore, performance simulation of emerging parallel applications is important for empirically evaluating innovations in a tightly coupled heterogeneous memory system.

Cycle-accurate simulation tools such as GPGPU-Sim [5] and gem5-gpu [6] are used today for evaluating CPU-GPU system performance. These simulation frameworks provide useful metrics such as execution time, network traffic, and energy consumed, but the massive parallelism of GPU cores makes it difficult to understand subtle changes to the memory system using only these coarse-grained measurements. When comparing the execution times of different memory system configurations, it can be difficult to determine whether an unexpected result is due to faulty assumptions about the impact of a change, the manifestation of an unexpected system effect, or a bug in the simulator.

To help understand the tradeoffs of different memory configurations in a tightly coupled heterogeneous system, we created GPU Stall Inspector (GSI). GSI is a GPU stall attribution tool for an integrated CPU-GPU simulator that enables a more detailed classification of memory stalls than existing methods currently provide. To demonstrate the power of our tool, we use it to analyze the sources of performance differences for two recent innovations to heterogeneous memory systems. First, we compare a hybrid, hardware-software coherence protocol for GPUs [7] to a conventional modern GPU coherence protocol. Next, we compare conventional scratchpad memories with two approaches that improve the performance of scratchpads [8], [9]. By analyzing the results of GSI and identifying the sources of memory stalls, we obtain a deeper understanding of the effects of these architectural innovations and motivate further changes to hardware and software.

II. Related Work

There have been previous efforts to provide more detailed performance profiling metrics. GPGPU-Sim is a cycle-accurate GPU simulator that can be used to model the effects of architectural changes to a GPU device [5]. It provides many simulation evaluation metrics including overall performance and stall counts, but not a stall breakdown. Aerialvision [10] augments GPGPU-Sim to provide more detailed measurements such as the number of cycles spent in the core pipeline by each thread, the number of DRAM accesses generated by each instruction, and the number of stall cycles due to long-latency off-chip read requests. However, it lacks a single comprehensive breakdown attributing execution latency to fine-grained stall causes.

O'Neil and Burtscher [11] use a stall classification method, also based on GPGPU-Sim, to help understand the performance of irregular GPU kernels. Their breakdown, however, does not differentiate between stalls due to memory or compute delays. Lee and Wu [12] also use a similar GPU stall classification method based on GPGPU-Sim which is designed to characterize the latency-hiding capability of GPU warp schedulers. It provides a more detailed stall breakdown and adds a memory latency stall type, but is focused on warp scheduler changes.

While this prior work also provides profiling information about GPUs, unlike GSI, they are all designed for discrete GPUs. Furthermore, they focus on profiling the GPU core, while we are capturing detailed information about the causes of memory stalls in a unified CPU-GPU memory system.

III. Background: GPU Execution

Without loss of generality, we use CUDA [13] terminology when describing GPUs. GPUs consist of multiple compute units called streaming multiprocessors (SMs) which are designed to efficiently execute thousands of concurrent threads by exploiting SIMD and SIMT parallelism. When a kernel is launched to a GPU, a scheduler begins assigning the specified number of threads to the SMs. The threads of a kernel are divided hierarchically into a grid of thread blocks, and thread blocks are further subdivided into warps. Thread block size defines SM scheduling granularity. All threads in a thread block are scheduled on the same SM and occupy that SM until they complete. Warps define pipeline scheduling granularity. All threads in a warp are issued and progress through the pipeline together.

The issue stage of an SM may consider only one instruction from each warp at any time. If the next instruction to be issued in a warp cannot issue because of a data or structural dependency, then the entire warp is blocked. An application with many long-latency memory or compute instructions can therefore experience reduced SM utilization and increased execution time. Similarly, an application that uses an execution unit in a bursty manner may incur underutilization and delay because many warps will be blocked waiting for a single hardware resource.

The warp scheduler may consider multiple warps in each cycle, and it may be able to issue an instruction from multiple warps in each cycle depending on the number of issue slots it has. The ability of GPUs to hide sources of delay with parallelism makes it very difficult to precisely understand the performance effects of application or architectural changes in a GPU.

IV. GSI Stall Classification

Understanding causes of GPU latency is complicated by the massive parallelism present in GPUs. In each cycle, a GPU may issue one instruction per issue slot from its warps. We define a stall cycle as any cycle in which no warp instructions are issued by an SM. In Section IV-A we describe the various stall types that may prevent a warp instruction from issuing. Although this work focuses on memory system stalls, we define and discuss the implications of all stall types to provide a full picture.

In each stall cycle there may be multiple different warp instructions that are stalled for multiple different reasons. In Section IV-B we describe how a single stall type is chosen from among these stall causes and attributed to each stall cycle. Finally, since we are most interested in memory stalls, in Sections IV-C and IV-D we subclassify memory stalls further based on their underlying causes.

A. Classifying Causes of Instruction Stalls

When an instruction can be issued in a cycle, the cycle is classified as **no stall**.

An **idle stall** occurs when there are no active warps available to issue instructions. A large number of idle stalls indicates that the kernel is not fully utilizing the GPU because some SMs simply have no work to do. This may be because the thread blocks are not evenly distributed across SMs or because thread block execution time is highly variable.

A **control stall** occurs when the instruction supplied by the instruction buffer is not the next instruction to be executed in a warp. If control stalls dominate, there is significant divergence in the kernel code.

A **synchronization stall** occurs when a warp is blocked due to a pending synchronization operation (acquire, release, or thread barrier).GPUs use acquire and release operations to preserve memory consistency and thread barriers to synchronize threads in a thread block. Acquire and release operations block a warp until one or more previous memory accesses have completed. A thread barrier blocks a warp until all other threads in the thread block have reached the barrier.

A **memory data stall** occurs when an instruction cannot issue because it is dependent on the output of a pending load. Section IV-C provides more details on the different subcategories of memory data stalls.

A **memory structural stall** occurs when a memory instruction is unable to issue to the load/store unit because it is full. There are multiple sources of memory structural stalls, which we describe in more detail in Section IV-D.

A **compute data stall** occurs when an instruction cannot issue because it is dependent on the output of a pending compute (non-memory) instruction.

A **compute structural stall** occurs when a compute instruction cannot issue because the appropriate compute unit is occupied.

B. Attributing Stalls to Cycles

GSI stall cycle attribution proceeds in two stages. First, a single stall type is assigned to each warp instruction considered in the issue stage. When classifying the stall cause assigned to an individual warp instruction, priority is given to the stall cause that is most strongly preventing execution. Intuitively, the strength of a stall cause is linked to the likelihood that the instruction will remain stalled in the next cycle. This is not always straightforward to define, and should be tailored to the profiling goals. We specify our prioritization in Algorithm 1.

After each instruction considered in the issue stage has been classified, the issue cycle itself is classified based on the stall causes of the individual warp instructions. The prioritization we use to classify cycle stall causes is defined in Algorithm 2. Generally, the cycle stall classification is set to match the weakest stall cause found. Intuitively, this is the stall cause of the instruction that was closest to issuing. Since we know it was the strongest stall cause for that instruction, removing that stall cause will likely improve performance. However, removing the stall cause may not cause a proportional reduction in the stall count for the removed stall type. A single stalled instruction can be blamed for many stall cycles even if all other instructions are unable to issue for unrelated reasons. Thus, removing the original stall cause and allowing this instruction to issue could simply cause other stall types to dominate.

The "weak" cycle stall classification priority described in Algorithm 2 is not an exact inversion of the "strong" instruction stall classification priority. Memory stalls and synchronization stalls are prioritized over compute stalls in both classification algorithms because we are interested in analyzing the effects of changes to the memory system.

C. Sub-Classifying Memory Data Stalls

Memory data stalls are subclassified based on where the dependency load was serviced. The sub-categories are: **L1 cache**, **L1 coalescing**, **L2 cache**, **remote L1 cache**, and **main memory**.

L1 cache stalls mean that instructions have been stalled because they are dependent on loads that are satisfied locally. This can happen if data is used immediately after it is loaded or if there is a delay in the load/store unit. L1 coalescing stalls are due to requests that have missed in the L1 cache but were satisfied by a response for another request

Algorithm 1 Instruction Stall Classification

```
if No active warps to consider then
    classify idle stall
else if The next instruction to issue is unavailable then
    classify control stall
else if Warp is blocked for a synchronization then
    classify synchronization stall
else if Instruction has a data hazard on a pending load then
    classify memory data stall
else if Instruction has a structural hazard on load/store unit then
    classify memory structural stall
else if Instruction has a data hazard on a pending compute operation then
    classify compute data stall
else if Instruction has a structural hazard on a compute unit then
    classify compute structural stall
else if Instruction is able to issue then
    classify no stall
end if
```

Algorithm 2 Issue Cycle Stall Classification

```
if At least one instruction was able to issue then
    classify no stall
else if At least one memory structural stall was found then
    classify memory structural stall
else if At least one memory data stall was found then
    classify memory data stall
else if At least one synchronization stall was found then
    classify synchronization stall
else if At least one compute structural stall was found then
    classify compute structural stall
else if At least one compute data stall was found then
    classify compute data stall
else if At least one control stall was found then
    classify control stall
else if At least one idle stall was found then
    classify idle stall
end if
```

to the same line. L2 cache stalls are due to dependencies for requests that are satisfied in the L2 and may mean that the L1 cache is not being efficiently utilized or that the data access pattern does not allow for reuse at L1. Remote L1 cache stalls are caused by dependencies for data requests that are satisfied at a remote L1 core. These are only possible in protocols like DeNovo[14] that enable ownership in L1 caches. Main memory stalls indicate dependencies on accesses to main memory and can occur when the data set is too large to fit in the L2 cache or the L2 is not being utilized efficiently.

D. Sub-Classifying Memory Structural Stalls

Memory structural stalls occur when a ready memory instruction is blocked from issuing to the load/store unit. Possible causes depend on the memory system being studied, but they are often due to multiple pending memory accesses. For the memory system configurations considered in this work, memory structural stalls can be caused by a **full MSHR**, a **full store buffer**, a **bank conflict**, a **pending release**, or a **pending DMA**.

Memory structural stalls due to a full MSHR may indicate there is bursty load miss traffic or the MSHR is too small. Similarly, stalls due to a full store buffer may mean there is bursty store miss traffic or the store buffer is too small. Bank conflict stalls can occur if data accesses are not evenly strided across cache or local memory banks. In the system studied, release operations block stores from issuing until all prior stores are flushed, causing pending release stalls. Similarly, a memory instruction will be blocked if it is trying to access DMA data before the DMA is complete, resulting in pending DMA stalls.

V. Methodology

Similar to the innovations we analyze in the case studies [7], [9], we simulate an integrated CPU-GPU system with a global, unified, shared address space. The system contains 1 CPU core and 15 GPU cores (SMs). The applications used in our first case study utilize all 15 GPU cores. The microbenchmark used in our second case study utilizes only one GPU core. CPU and GPU cores are uniformly distributed across a 4x4 mesh network. Each core has a private L1 and all cores share a banked last level L2 cache. Table I summarizes the key architectural parameters of our simulated system.

We simulate a tightly coupled CPU-GPU system using an integrated, cycle accurate architectural simulator. The Simics [15] full system simulator models the CPU cores, GPGPU-Sim v3.2.1 models the GPU cores (our GPU is similar to an NVIDIA GTX 480), the Wisconsin GEMS memory timing simulator [16] models the memory system, and Garnet [17] models the interconnect. The changes

CPU Parameters	
Frequency	2 GHz
Cores	1
GPU Parameters	
Frequency	700 MHz
SMs used: case study 1, case study 2	15, 1
Scratchpad/Stash Size	16 KB
Number of Banks in Stash/Scratchpad	32
Memory Hierarchy Parameters	
L1 and Stash/Scratchpad hit latency	1 cycle
Remote L1 and Stash hit latency	35−83 cycles
L1 Size (8 banks, 8-way assoc.)	32 KB
L2 Size (16 banks, NUCA)	4 MB
L2 hit latency	29−61 cycles
Memory latency	197−261 cycles

TABLE I: Parameters of the simulated heterogeneous system.

required to implement GPU stall profiling are minimal. GSI increases simulation time by on average 5% for a set of representative workloads, with this proportion decreasing as input sizes increase. We use atomic operations to perform global synchronization and all atomic accesses occur at L2. [1] We also use a data-race-free memory consistency model [19].

All memory configurations tested use a 32-entry MSHR and a 32-entry write-combining store buffer. The store buffer keeps track of pending writes and is flushed when it becomes full, at the end of a kernel, and on a release operation. By keeping track of which data in the cache is dirty, the store buffer enables write-combining and non-blocking stores for both coherence protocols studied, similar to other recent work [20].

VI. Case Study 1: DeNovo vs. GPU Coherence

Our first case study compares two coherence protocols: a baseline GPU coherence and the DeNovo coherence protocol. This comparison mirrors the protocol comparison performed by Sinclair, et al. in previous work [7]. By analyzing the stall breakdowns of these two coherence protocols for a single application, unbalanced tree search, we demonstrate how the information provided by GSI can enable a deeper understanding of performance results and can motivate changes to an application.

A. DeNovo vs. GPU Coherence Overview

Our baseline GPU coherence protocol, which we refer to as GPU coherence, is a simple software-based protocol similar to those of modern GPUs. Unlike MESI-style protocols, modern GPU coherence protocols use reader-initiated invalidations. Thus, an acquire operation (e.g., atomic read or kernel launch) invalidates the entire cache to

[1]This is different from the more optimized recent implementations of DeNovo which employ ownership even for atomics, enabling owned atomics to be serviced at the L1 [7], [18]

prevent subsequent reads from using stale values. Similarly, a release synchronization (e.g., an atomic write or kernel exit) flushes all buffered writes from the L1 caches. Instead of obtaining ownership on writes, GPU coherence protocols write-through dirty data to the shared L2 cache. As a result, read misses can obtain the up-to-date copy from the L2 cache. This coherence strategy is very simple and works well for conventional streaming GPU applications which only synchronize at coarse-grained kernel boundaries. However, for emerging applications with frequent synchronization or irregular data accesses [21], [22], [23], [24], [25], [26], GPU coherence can lead to poor cache utilization. Frequent acquire and release operations prevent cache reuse and write coalescing, reducing the effectiveness of the L1 cache.

Previous work demonstrated that DeNovo provides advantages over conventional GPU coherence protocols for GPUs [7] and conventional hardware coherence protocols like MESI for CPUs [14], [18], [27]. DeNovo keeps caches coherent by self-invalidating caches on acquire operations and registering for ownership of dirty data on release operations. An SM obtains ownership of written data by sending an ownership request to the last level cache, which keeps track of the owner of the up-to-date data. This data stays owned in the cache until it is evicted or until ownership is requested by another SM. Owned data is not invalidated on acquires or written back on releases, enabling improved cache performance in the presence of frequent synchronization. In both configurations studied, the CPU cache uses DeNovo coherence; we only compare DeNovo and GPU coherence protocols in the GPU caches. Both protocols self-invalidate on acquire operations and flush their store buffer on release operations.

The main difference between DeNovo and GPU coherence is DeNovo's ability to keep data owned in the cache. Ownership allows improved cache utilization, but it adds some overhead to the coherence protocol. If the last level cache receives a request for data that is owned at a remote core, it forwards the request to the owner which will respond directly to the requestor. This extra hop adds latency to remote read requests and ownership requests.

With multiple potentially competing effects, it can be difficult to discern which characteristics of DeNovo and ownership have the greatest performance impact using only coarse-grained simulation metrics. We compare the performance of GPU coherence and DeNovo coherence for a single benchmark application and we use GSI, detailed in section IV, to analyze how the protocol differences contribute to different types of stalls.

B. Unbalanced Tree Search Benchmark

Unbalanced tree search (UTS) is an algorithm that processes each node of an unbalanced tree with unknown structure [28]. The control and access pattern is irregular and a global task queue is used to keep track of which nodes have yet to be processed. Each element in the task queue corresponds to a tree node. When a tree node is processed, its children are pushed onto the global task queue. Threads are grouped into warps which follow the same control path. Task queue access is protected by a lock that is only accessed by one thread per warp. This lock ensures that only one warp may pull from or push to the queue at any time. The lock is implemented using atomic instructions with acquire and release semantics to properly synchronize concurrent access.

This application is representative of programs with high levels of irregular synchronization (e.g. task queue based algorithms). Although such algorithms are not commonly used on GPUs today due to the inefficiencies of synchronization on modern GPUs, this data-driven approach has been shown to speed up certain types of irregular GPU applications [29] and may become more common as GPUs become more general purpose and GPU synchronization improves.

C. Unbalanced Tree Search Results

Figure 1a shows the GPU execution time breakdown of GPU coherence and DeNovo running the UTS benchmark. Figures 1b and 1c show the memory data and structural stall subclassifications, respectively. We use these results to identify sources of performance degradation in the two executions.

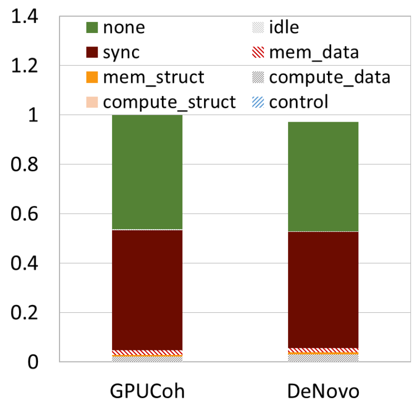
As Figure 1a shows, there is very little overall performance difference between GPU coherence and DeNovo coherence for UTS. This is because the execution time for both is dominated by synchronization stalls. The large number of synchronization stalls are due to the use of a single global lock for the global task queue. All workers must acquire and release this lock before processing a node and will spin on the shared variable until the lock is free.

The stall breakdown identifies synchronization as the most prominent source of stalls for both systems. Therefore, any efforts to improve performance must reduce synchronization stalls either through changes to the application or changes to the system.

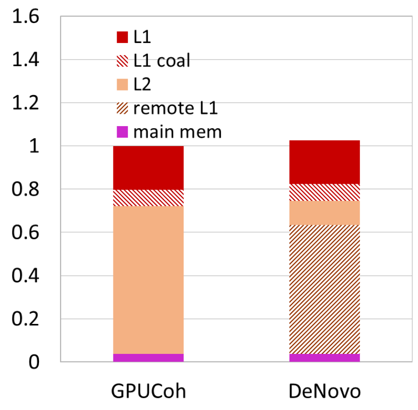
D. Unbalanced Tree Search Decentralized

One way to reduce synchronization in the application is to decentralize the task queue. If each SM has its own task queue to push to and pull from, contention for the queue locks will be much lower and workers will be able to make forward progress more easily.

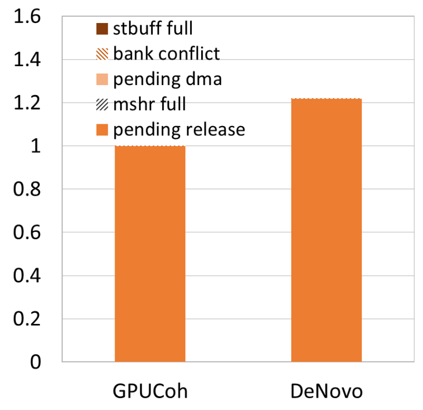
We modified UTS to implement this change and refer to this version as unbalanced tree search decentralized (UTSD). Similar to prior work, UTSD adds per-SM local task queues to UTS and allows each worker to push and pull from its local queue [23]. Load balancing is preserved by adding a shared global queue. Threads only push to the



(a) execution time breakdown

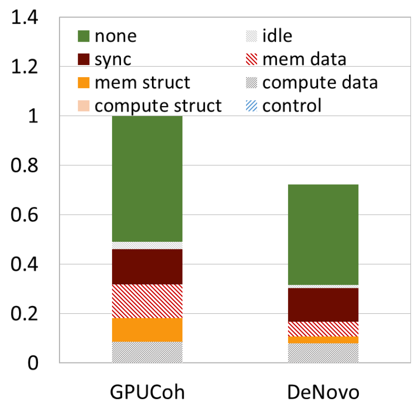


(b) memory data stall breakdown

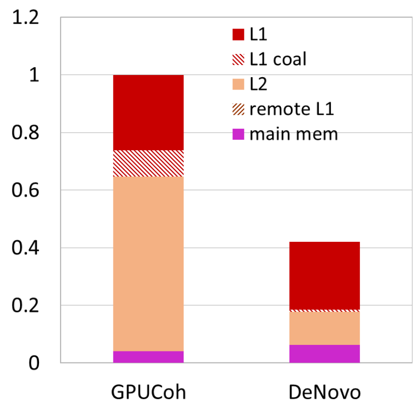


(c) memory structural stall breakdown

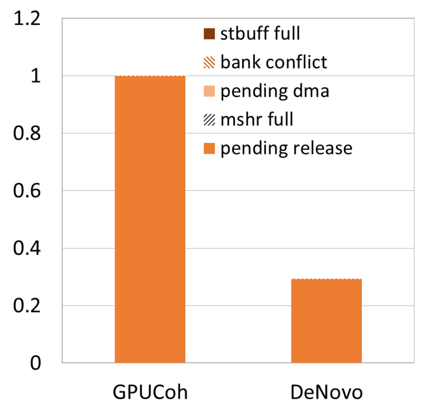
Fig. 1: Stall cycle breakdowns for UTS (normalized to GPU coherence). For this version of UTS, memory time is small, so the last two sub-figures are shown primarily for completeness.



(a) execution time breakdown



(b) memory data stall breakdown



(c) memory structural stall breakdown

Fig. 2: Stall cycle breakdowns for UTSD (normalized to GPU coherence)

global queue when the local queue is full and only pull from the global queue when the local queue is empty. Local queues also increase cache locality, since task producers and consumers are likely to be on the same SM.

Figure 2a shows the stall breakdown for GPU coherence and DeNovo coherence running UTSD, normalized to GPU coherence. Although it is not visible in the normalized graphs, UTSD reduces the execution time for GPU coherence and DeNovo coherence by 91% and 94%, respectively, relative to UTS. This decrease comes from a significant reduction in synchronization stalls and "no stall" cycles, which occur due to reduced lock contention.[2] In addition, UTSD increases the relative proportion of memory stalls, highlighting the memory system tradeoffs of the two coherence protocols.

DeNovo coherence decreases memory structural stalls by 71% and memory data stalls by 57% relative to GPU coherence, contributing to a 28% reduction in execution time. Figure 2b shows that the memory data stall reduction is primarily due to a decrease in requests serviced at L2. The main memory and L1 cache components of memory data stalls are not reduced because DeNovo's increased hit rate affects data that has a high locality and would otherwise be located in the L2. Thus stalls due to requests that are serviced at L1 or main memory for GPU coherence do not change under DeNovo. Figure 2c shows that memory structural stalls are reduced because pending release stalls decrease. These improvements demonstrate how DeNovo improves performance by obtaining ownership in the local L1 caches. Since owned data is not invalidated on acquire operations, DeNovo is able to decrease delay by reusing cached L1 data across synchronization points. Similarly, release operations are cheaper for DeNovo because owned data does not need to be written back on a release, so DeNovo incurs fewer stalls due to pending release operations. The significant decreases in both memory data and

[2]"No stall" cycles also decrease because fewer instructions are executed spinning on a lock.

memory structural stalls therefore give us a measure of how much ownership can reduce latency by improving cache reuse and reducing the cost of synchronization in the UTSD benchmark.

The GSI stall breakdowns indicate that DeNovo's disadvantages do not have a significant impact on UTSD's latency. If the overheads of ownership were significant and the redirection of load requests and ownership requests caused delay, we would expect to see specific increases in memory stalls. Load request redirection would cause increased memory data stalls due to remote L1 hits. Ownership request redirection would increase store buffer flush latency and cause increased memory structural stalls due to pending release operations. In fact, we do see these effects in the memory stall breakdowns for the original UTS benchmark. These effects indicate that redirection is fairly common in UTS, meaning locality is poor and most tasks are likely consumed on a different node than where they were produced. With UTSD's local task queues, memory data stalls due to remote L1 hits virtually disappear and memory structural stalls due to pending releases are reduced relative to GPU coherence, indicating that UTSD's locality removes the disadvantages of ownership.

The stall profiling information can also be used to motivate and prioritize further application optimizations. Synchronization stalls, for example, still contribute significantly to execution time. This suggests that additional performance gains can be achieved by reducing synchronization costs even further. Synchronization may be reduced in the application by adding backoff or by grouping threads at a coarser granularity (e.g. thread blocks rather than warps). Hardware optimizations such as owned atomics [7] or scoped atomics [23] can also make synchronization cheaper by performing atomic accesses locally. Finally, UTSD indicates that memory structural stalls due to a pending release contribute to 10% of execution time for GPU coherence and 4% of execution time for DeNovo. This latency can be reduced or eliminated if we add a mechanism similar to QuickRelease's S-FIFO [20]. By keeping track of the stores that were ordered before each release operation, the S-FIFO allows memory requests to continue to issue while a release is in progress.

In this case study we have shown how GSI can provide information that motivates targeted optimizations to hardware and software. We enacted one such software optimization to the UTS benchmark and witnessed substantial performance improvements for both GPU coherence and DeNovo. GSI helped us analyze and quantify the performance differences between the two configurations, and between the original application and the improved application. Finally, we used the stall profiling results of the improved application to evaluate the likely effects of additional performance optimizations.

VII. Case Study 2: Stash vs. Scratchpad+DMA

For our second case study we evaluate two improvements to the scratchpad memory structure used in modern GPUs. We compare the performance of these innovations against a baseline scratchpad implementation and analyze the stall breakdowns of these two innovations for a single application. Using this information we again determine the precise causes of performance degradation and motivate changes to the hardware configuration.

A. Stash vs Scratchpad+DMA Overview

Scratchpad memory is a directly addressed memory space that is not kept coherent and is private to a thread block. The scratchpad can be used to cheaply satisfy data accesses that would otherwise need to use the cache. Because all threads in a thread block share the space, scratchpad memory is often used for efficient intra-thread block communication as well.

Despite these benefits, scratchpads suffer from multiple inefficiencies. Typical scratchpad use involves loading data into the scratchpad at the start of a kernel, performing computations on the data in the scratchpad, and then copying the scratchpad data to global memory at the end of the kernel. In this scenario, copying data between the global memory and the scratchpad pollutes the cache and registers and increases instruction count and energy. In addition, the lack of coherence requires that any data that could potentially be accessed by other cores must be conservatively transferred to and from the scratchpad. Thus, if a memory region is only accessed sparsely and unpredictably, the entire region must be copied to and from the scratchpad memory, incurring waste. Because of these inefficiencies, only kernels that access a memory region multiple times in a predictable fashion tend to benefit from using scratchpads.

D2MA [8] addresses the inefficiencies of scratchpads by offloading the process of loading data into the scratchpad to a separate DMA. The DMA engine explicitly transfers data into the scratchpad in bulk without polluting the L1 cache or registers. At the start of a kernel a mapping from scratchpad to global data is defined. Scratchpad memory accesses to the mapped region are then blocked (on the first use) until the entire DMA transfer is complete. We approximate D2MA by adding a DMA engine to the baseline scratchpad memory. This implementation, which we refer to as scratchpad+DMA, differs from D2MA in that the DMA engine can transfer data to and from scratchpad (D2MA only loads data into the scratchpad) and a load to an incomplete DMA mapping blocks progress at a core granularity rather than a warp granularity.

Stash is a new hybrid memory structure that addresses the inefficiencies of scratchpad accesses and cache accesses by making scratchpads a part of the coherent global address space [9]. The stash stores a mapping from local to global data (and the reverse) in a stash map structure. When any address in the stash mapping is first accessed, the stash map is used to generate a global request to the memory system. When the data returns, it bypasses the cache and is directly loaded into the stash. Subsequent accesses to the mapped address will always hit locally and return without translation. The stash map also enables an SM to respond to global requests for data that is dirty in the stash. These coherence mechanisms allow stash data to be loaded on-demand and to be lazily written back, which is not possible with D2MA. This avoids wasteful copies and wasteful instructions involved in managing a scratchpad. Our stash implementation is the same as the one used by Komuravelli, et al. [9].

All three configurations use DeNovo coherence for GPU and CPU caches and for the stash. Both scratchpad+DMA and stash improve upon scratchpad memory because the data transfers to and from scratchpad/stash memory bypass the cache. The stash also implicitly transfers data between the local and global address spaces. The main difference between stash and scratchpad+DMA is that stash data is kept coherent with the shared global memory. This means that stash accesses can generate global load requests on-demand and that dirty data can be lazily written back because it is globally visible. In contrast, scratchpad with DMA must conservatively transfer all data into the scratchpad at the start of a kernel, and conservatively write back any potentially modified data at the end of the kernel.

The DMA data movement pattern can be helpful or harmful. Bulk reads and writes happen in parallel and can be a very effective preloading technique for avoiding the memory latency of successive on-demand load misses. However, scratchpad accesses to a pending DMA allocation must stall until the entire DMA transfer completes. This latency combined with the congestion caused by bursty DMA traffic may negatively affect performance.

The tradeoffs of stash memory and scratchpad+DMA result in differences in the memory system which can reduce some sources of performance degradation but increase others. We use GSI to better understand how these differences affect GPU performance, focusing on one microbenchmark.

B. Implicit Microbenchmark

Implicit is one of the synthetic microbenchmarks used by Komuravelli, et al. to evaluate the benefits of stash [9]. In the implicit microbenchmark, an array of data is allocated and mapped to scratchpad/stash memory. Each thread block is assigned a chunk of the array. Each thread reads an element in the chunk, performs a computation on the element, and writes the result back to the same location in memory.

This microbenchmark highlights the advantage of implicitly loading data into scratchpad/stash memory and is representative of applications that access data in a regular streaming manner. Stash implicitly moves data from global to scratchpad/stash memory while scratchpad+DMA uses a DMA engine to reduce the overheads of explicit data movement, so both configurations should improve performance over the baseline scratchpad configuration. By applying GSI to this application and analyzing the stall breakdown, we can understand how important each of the above effects are on overall performance, and we use this information to motivate a change in the hardware system used.

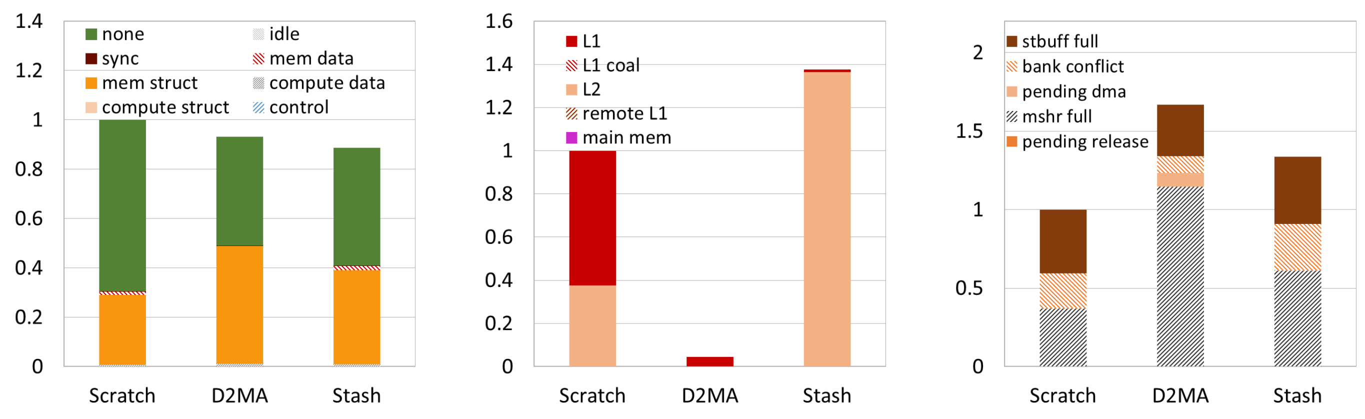
C. Implicit Results

Figure 3a shows the GPU execution time breakdown of the implicit microbenchmark. Figures 3b and 3c show the memory data stall and memory structural stall subclassifications, respectively. We use these results to identify causes of performance disparity between stash and scratchpad+DMA memory configurations.

Figure 3a shows that the number of "no stall" cycles is reduced by 36% and 31% for scratchpad+DMA and stash, respectively. This occurs because both the scratchpad+DMA and stash configurations reduce instruction count. The scratchpad+DMA configuration does this by offloading the loads to its DMA engine, and the stash does this by loading mapped data on-demand from global memory into the stash. Thus, both configurations do not need initial scratchpad loads and store instructions that pollute the registers or L1 cache. However, the reduction in "no stall" cycles is offset by increases in other stall types. This kind of benefit offset is not uncommon when profiling stall causes. A GPU core may be unable to issue instructions for several reasons, but the stall is only attributed to one cause.

In the case of the implicit microbenchmark, the "no stall" cycle reductions in scratchpad+DMA and stash are offset primarily by a 67% and 34% increase in memory structural stalls, respectively. Scratchpad has fewer memory structural stalls because the explicit transfer of scratchpad data serves to restrict the rate of requests to the memory system. At the start and end of the kernel, the baseline scratchpad implementation issues many concurrent loads and stores (respectively) to the memory system, incurring memory structural stalls due to bank conflicts, a full store buffer and a full MSHR. However, these memory operations are interleaved with instructions that perform address calculations and scratchpad store instructions, so the rate at which global memory loads are issued to the memory system is limited. As a result, the number of memory structural stalls is also limited.

In comparison, scratchpad+DMA does not need to process any data transfer instructions and is able to issue load requests to the memory system as fast as the DMA engine allows (one per cycle). This increased request rate causes the MSHR to fill up faster. These generated requests bypass the pipeline and the cache, so bank conflicts are insignificant, but as soon as a normal memory access tries to issue to the global memory or scratchpad memory, it is blocked due



(a) execution time breakdown (b) memory data stall breakdown (c) memory structural stall breakdown

Fig. 3: Stall cycle breakdowns for implicit microbenchmark (normalized to baseline scratchpad)

to a full MSHR or a pending DMA (only for dependent instructions).

Stash generates a global memory access at the first load of each stash data word. Since the stash is directly addressed, fewer instructions are needed to compute stash addresses than are needed to compute the target address for requests to global memory in the baseline scratchpad configuration. This means fewer instructions are interleaved with the memory instructions, and memory instructions are issued to the memory system at a faster rate than for baseline scratchpad. This increased memory request rate increases the amount of memory structural stalls due to a full MSHR, a full store buffer, and bank conflicts.

Overall, the stall breakdowns tell us that scratchpad+DMA and stash are able to improve performance, but the amount of time saved is offset by additional stalls caused by increased memory request frequency. Furthermore, although there is a difference in data stalls, it is insignificant.

D. Implicit MSHR Sensitivity

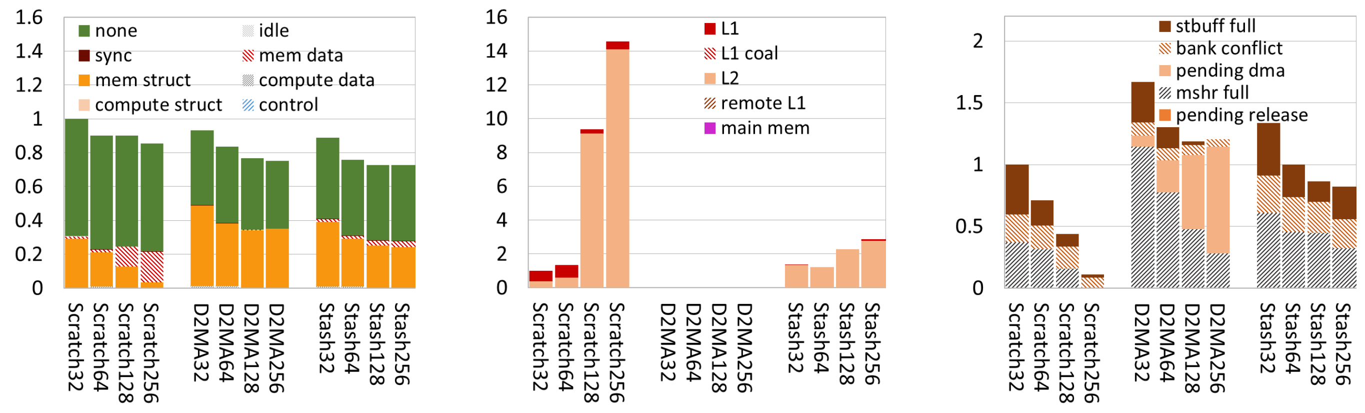
Since MSHR stalls are significant for this microbenchmark, we investigate the effect of increasing MSHR size on performance. We run the implicit microbenchmark on all three configurations while increasing MSHR size from 32 to 256. We also scale the store buffer size with the MSHR size to prevent store buffer stalls from becoming the new bottleneck. Figure 4a shows the results normalized to baseline scratchpad using an MSHR with 32 entries.

The decrease in full MSHR stalls shows that increasing MSHR size benefits all configurations. However, each configuration benefits from the increased MSHR size in different ways. A 256-entry MSHR completely eliminates full MSHR stalls for the baseline scratchpad configuration, but memory data stalls significantly increase (13X compared to its original value). Memory data stalls increase for the scratchpad configurations because the instruction following the explicit load is dependent on the result of that load (a store of the value to scratchpad memory). Although many more explicit loads are able to issue with a larger MSHR, the next instruction will need to block until the load has completed.

As MSHR size increases, MSHR stalls decrease for the scratchpad+DMA configuration. However, memory structural stalls due to pending DMA requests significantly increase as MSHR size increases (8.9X more with a 256-entry MSHR). These memory structural stalls increase because the first scratchpad accesses occur early in the application, which causes threads to be blocked due to a pending DMA soon after the application begins.

Increasing MSHR size also reduces MSHR stalls for the stash configuration by allowing a greater number of threads to concurrently issue memory requests (implicitly in the case of the stash configuration). Although MSHR stalls decrease, memory data stalls increase (2.1X compared to the original value) because the issued requests have not completed before a dependent instruction needs to use the returned value. However, the increase is less significant for stash compared with scratchpad. This is because the scratchpad implementation must wait for all loads to the scratchpad to complete before proceeding to the compute phase. The stash configuration, on the other hand, will issue loads on-demand within the compute phase. Although many warps will still be blocked waiting for the result of a load to stash memory, there are more active threads during the compute phase that can utilize the issue slots with useful work. As a result, stash achieves higher core utilization than scratchpad and experiences a smaller increase in memory data stalls as MSHR size increases.

Overall, these results show that increasing MSHR size improves performance for all configurations. However, alleviating this bottleneck causes other stall sources to increase. For the scratchpad configuration, increasing the number of



(a) execution time breakdown (b) memory data stall breakdown (c) memory structural stall breakdown

Fig. 4: Stall cycle breakdowns for implicit microbenchmark with varying MSHR sizes (normalized to baseline scratchpad with 32-entry MSHR)

MSHRs does not help as much because of data dependencies. Increasing MSHR size is more beneficial for the scratchpad+DMA configuration, which sidesteps the data dependencies by prefetching data. However, because the scratchpad data is used early in the application, memory structural stalls due to pending DMA requests increase for the scratchpad+DMA configuration. The stash configuration also benefits from increased MSHR size but is better able to utilize the core by only blocking scratchpad loads at the granularity of warps (as discussed in Section VII-A). Thus, these results demonstrate the benefits of the stash's hybrid memory organization.

VIII. Conclusion

If heterogeneity and GPGPUs in particular are to continue to drive performance improvements in the coming years, then significant inefficiencies must be overcome to enable their use on a wider range of parallel applications. Current CPU-GPU memory systems are one significant source of inefficiency. As innovations in heterogeneous memory systems emerge to address these issues, better techniques are needed for analyzing their effects on performance.

We propose GPU Stall Inspector, a new stall profiling system that provides detailed information about sources of stalls in a tightly coupled GPU. Each stall cycle is classified based on its cause. Memory stalls are subclassified further based on where the memory request was serviced or based on the cause of delay in the load/store pipeline. By identifying and quantifying the effects of GPU latency sources, GSI can provide valuable insight when evaluating a wide range of software and hardware innovations in highly parallel heterogeneous systems.

We present two case studies which demonstrate the value of detailed GPU stall profiling. Applied to UTS and the implicit microbenchmark, the stall breakdown reveals differences in synchronization and memory stall sources which help motivate software and hardware changes.

Moreover, although we focus primarily on memory stalls, GSI can easily be applied to study software and architectural changes which predominantly affect other stall sources. For example, when studying architectural changes that affect functional unit congestion or latency, compute stalls may be prioritized or subcategorized instead of memory stalls. Similarly, when studying software changes that affect control divergence, control stalls may be prioritized or subcategorized. Thus, GSI's fundamental methodology may be useful in an even broader range of application domains.

Acknowledgments

This work was supported in part by the Center for Future Architectures Research (C-FAR), one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, the National Science Foundation under grants CCF-1018796 and CCF-1302641, and a Qualcomm Innovation Fellowship for Sinclair and Komuravelli.

References

[1] A. Branover, D. Foley, and M. Steinman, "AMD Fusion APU: Llano," *IEEE Micro*, vol. 32, no. 2, pp. 28–37, 2012.
[2] IntelPR, "Intel Delivers New Range of Developer Tools for Gaming, Media," *Intel Newsroom*, 2013.
[3] P. R. (AMD), "OpenCL 2.0 - Shared Virtual Memory," *AMD Developer Central*, 2014.
[4] P. Rogers, "Heterogeneous System Architecture Overview," in *Hot Chips*, 2013.
[5] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, 2009.

[6] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, “gem5-gpu: A Heterogeneous CPU-GPU Simulator,” *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[7] M. D. Sinclair, J. Alsop, and S. V. Adve, “Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models,” in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 647–659, 2015.

[8] D. A. Jamshidi, M. Samadi, and S. Mahlke, “D2MA: Accelerating Coarse-grained Data Transfer for GPUs,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 431–442, 2014.

[9] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, P. Srivastava, M. Kotsifakou, S. V. Adve, and V. S. Adve, “Stash: Have Your Scratchpad and Cache it Too,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 707–719, 2015.

[10] A. Ariel, W. Fung, A. Turner, and T. Aamodt, “Visualizing Complex Dynamics in Many-Core Accelerator Architectures,” in *IEEE International Symposium on Performance Analysis of Systems Software*, pp. 164–174, 2010.

[11] M. A. O’Neil and M. Burtscher, “Microarchitectural Performance Characterization of Irregular GPU kernels,” in *IEEE International Symposium on Workload Characterization*, pp. 130–139, 2014.

[12] S.-Y. Lee and C.-J. Wu, “CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, pp. 175–186, 2014.

[13] NVIDIA, “CUDA SDK 3.1.” http://developer.nvidia.com/object/cuda_3_1_downloads.html.

[14] H. Sung, R. Komuravelli, and S. V. Adve, “DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 13–26, 2013.

[15] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A Full System Simulation Platform,” *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[16] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, “Multifacet’s General Execution-Driven Multiprocessor Simulator (GEMS) Toolset,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[17] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, “GARNET: A detailed on-chip network model inside a full-system simulator,” in *IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 33–42, 2009.

[18] H. Sung and S. V. Adve, “DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations,” in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 545–559, 2015.

[19] S. Adve and M. Hill, “Weak Ordering – A New Definition,” in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA, 1990.

[20] B. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, D. Wood, *et al.*, “QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs,” in *IEEE 20th International Symposium on High Performance Computer Architecture*, pp. 189–200, 2014.

[21] J. A. Stuart and J. D. Owens, “Efficient Synchronization Primitives for GPUs,” *CoRR*, vol. abs/1110.4623, 2011.

[22] M. Burtscher, R. Nasre, and K. Pingali, “A Quantitative Study of Irregular Programs on GPUs,” in *2012 IEEE International Symposium on Workload Characterization*, pp. 141–151, 2012.

[23] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, “Heterogeneous-Race-Free Memory Models,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 427–440, 2014.

[24] J. Y. Kim and C. Batten, “Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists,” in *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 75–87, 2014.

[25] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron, “Pannotia: Understanding Irregular GPGPU graph applications,” in *IEEE International Symposium on Workload Characterization*, pp. 185–195, 2013.

[26] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood, “Synchronization Using Remote-Scope Promotion,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 73–86, 2015.

[27] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, “Denovo: Rethinking the Memory Hierarchy for Disciplined Parallelism,” in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pp. 155–166, 2011.

[28] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, “UTS: An Unbalanced Tree Search Benchmark,” in *Languages and Compilers for Parallel Computing*, vol. 4382 of *Lecture Notes in Computer Science*, pp. 235–250, Springer Berlin Heidelberg, 2007.

[29] J. Y. Kim and C. Batten, “Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists,” in *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 75–87, 2014.