Inter-kernel Reuse-aware Thread Block Scheduling

MUHAMMAD HUZAIFA, University of Illinois at Urbana-Champaign, USA
JOHNATHAN ALSOP, AMD Research, USA
ABDULRAHMAN MAHMOUD, University of Illinois at Urbana-Champaign, USA
GIORDANO SALVADOR, Unaffiliated
MATTHEW D. SINCLAIR, University of Wisconsin-Madison, USA and AMD Research, USA
SARITA V. ADVE, University of Illinois at Urbana-Champaign, USA

As GPUs have become more programmable, their performance and energy benefits have made them increasingly popular. However, while GPU compute units continue to improve in performance, on-chip memories lag behind and data accesses are becoming increasingly expensive in performance and energy. Emerging GPU coherence protocols can mitigate this bottleneck by exploiting data reuse in GPU caches across kernel boundaries. Unfortunately, current GPU thread block schedulers are typically not designed to expose such reuse. This article proposes new hardware thread block schedulers that optimize inter-kernel reuse while using work stealing to preserve load balance. Our schedulers are simple, decentralized, and have extremely low overhead. Compared to a baseline round-robin scheduler, the best performing scheduler reduces average execution time and energy by 19% and 11%, respectively, in regular applications, and 10% and 8%, respectively, in irregular applications.

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**; **Parallel architectures**;

Additional Key Words and Phrases: GPUs, memory systems, scheduling, caches

ACM Reference format:
Muhammad Huzaifa, Johnathan Alsop, Abdulrahman Mahmoud, Giordano Salvador, Matthew D. Sinclair, and Sarita V. Adve. 2020. Inter-kernel Reuse-aware Thread Block Scheduling. *ACM Trans. Archit. Code Optim.* 17, 3, Article 24 (August 2020), 27 pages.
https://doi.org/10.1145/3406538

This work was supported in part by a Sohaib and Sara Abbasi Computer Science Fellowship for Huzaifa, the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA, 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 13-02641 and CCF 16-19245, and by a Google Faculty Research Award.
Authors' addresses: M. Huzaifa, A. Mahmoud, and S. V. Adve, University of Illinois at Urbana-Champaign, Department of Computer Science, 201 N. Goodwin Ave., Urbana, IL, 61801; emails: {huzaifa2, amahmou2, sadve}@illinois.edu; J. Alsop, AMD Research, 2002 156th Ave. NE Suite 300, Bellevue, WA, 98007; email: johnathan.alsop@amd.com; G. Salvador, Unaffiliated; email: cs.giordano.salvador@gmail.com; M. D. Sinclair, University of Wisconsin-Madison, Computer Sciences Department, 1210 W. Dayton St., Madison, WI, 53706, AMD Research, 2002 - 156th Ave. NE Suite 300, Bellevue, WA, 98007; email: sinclair@cs.wisc.edu.

1544-3566/2020/08-ART24
https://doi.org/10.1145/3406538

1 INTRODUCTION

With the end of Dennard scaling, modern systems are increasingly using specialized energy-efficient accelerators. GPUs are among the most popular accelerators today. Although GPUs implement deeply pipelined memory hierarchies to satisfy the memory requests of thousands of threads, there is a growing gap between the computational and memory capabilities of GPUs—memory bandwidth has begun to stagnate and the memory hierarchy has become a major consumer of energy [21, 56]. Continued scaling of performance and energy efficiency is, therefore, contingent upon efficient utilization of the memory hierarchy.

Although GPUs are now equipped with multi-level cache hierarchies, they are typically poorly utilized [9] and are unable to bridge the compute-memory gap. One source of locality that is not exploited by current GPU caches is *inter-kernel locality*. Many GPU applications execute multiple kernels that operate on the same data. By reusing these data across kernels; i.e., by exploiting *inter-kernel data reuse*, required data can be fetched from the cache instead of main memory. This can reduce memory pressure, improve energy efficiency, and increase performance. However, current GPUs are *unable* to exploit inter-kernel locality due to the inability of modern GPU coherence protocols to keep data cached across kernel boundaries.

Current GPU coherence protocols are simple, software-driven, and rely on data-race-freedom [41]. At a synchronization point, a core writes through all its dirty data to the shared last-level cache; it also invalidates all data in its private cache (typically, L1) to ensure it does not access any stale data in the future.[1] Since kernel boundaries are synchronization points, inter-kernel reuse at the L1 cannot be exploited as the data have been invalidated. However, several emerging GPU coherence protocols [4, 25, 41] do not (completely) invalidate L1 caches at synchronization points. We design new thread block (TB) schedulers that exploit these protocols to increase inter-kernel reuse, thereby improving performance and energy.

We design four new decentralized hardware TB schedulers that seek to exploit inter-kernel reuse (locality) by scheduling threads that require a piece of data on the same core as threads (from a previous kernel) that produced or shared that data. Although the focus of our schedulers is exploiting inter-kernel reuse, they also inherently exploit some intra-kernel reuse as a side-effect (described in Section 4).

Our schedulers utilize commonly observed producer-consumer relationships of TBs from different kernels to increase data reuse with very low overhead. The first three schedulers ignore load-balancing and focus only on cache reuse. However, irregular applications such as graph analytics necessitate balancing the tension between load-balance and cache reuse. Thus, the fourth, most advanced scheduler employs hardware work stealing to balance work across the entire GPU while still enabling inter-kernel reuse.

While there has been previous work that exploits inter-kernel reuse for GPU applications [51], it focuses on nested parent-child kernels, schedules kernels instead of TBs, and has to perform expensive flushes to ensure child kernels do not read stale data. In contrast, our work is able to exploit inter-kernel reuse across arbitrary kernels and schedules TBs rather than kernels. There is also prior work that increases intra-kernel reuse [8, 26, 29, 49]—such reuse is an added benefit of our work, but not its main focus. Section 9 provides a more comprehensive discussion of related work.

Overall, we make the following contributions: We propose four new inter-kernel reuse-aware (IKRA) schedulers that are simple and decentralized, and we use the inherent nature of common TB dependencies to exploit inter-kernel data reuse. We evaluate these schedulers across a variety of

[1]Without loss of generality, in this article, we assume a private L1 per GPU core and an L2 shared across all GPU cores. Our ideas apply to other hierarchies with multiple private cache levels as well.

microbenchmarks and applications, relative to a round-robin baseline scheduler. For eight regular, iterative applications, we find our best-performing scheduler decreases average execution time, energy, and network traffic by 19% (max 43%), 11% (max 21%), and 42% (max 86%), respectively. We find that modern, regular GPU applications are inherently load-balanced, and hardware work stealing only marginally improves performance. In contrast, evaluating four irregular, iterative GPU applications on two inputs each, we show that the scheduler with reuse-aware work stealing performs best—compared to round-robin, it reduces execution time by 10% (max 18%), energy by 8% (max 19%), and network traffic by 16% (max 37%). We also perform several sensitivity studies on the IKRA schedulers. We evaluate them on a range of L1 cache sizes and show that in some cases, they are able to provide benefits even with L1 sizes as low as 16 KB. We evaluate the schedulers for a system that only provides coherence at the L2, thereby forgoing any opportunities for inter-kernel reuse at the L1 (as with conventional GPUs). We show that while the IKRA schedulers show some benefits for such systems due to L2-only reuse, a significant source of benefits is L1 reuse, motivating the newer coherence protocols that enable such reuse. Finally, we also evaluate our schedulers in application scenarios with no inter-kernel reuse to understand any adverse effects on performance. We find little to no impact (average 4% improvement due to intra-kernel reuse).

2 BACKGROUND

2.1 Cache Coherence for GPUs

GPUs generally use simple, software-based coherence protocols that rely on data-race-freedom [41]. At a synchronization point, a GPU core writes-through all its dirty data to the shared last-level cache and invalidates all data in its private (L1) cache. This ensures that no stale data are accessed after the synchronization point. We call this coherence protocol GPU Coherence. This scheme, however, precludes inter-kernel reuse, as the (L1) cache is invalidated at kernel boundaries. Although conventional multicore CPU coherence protocols like MESI can exploit reuse across kernel boundaries, they have been shown to not be a good fit for GPUs [14, 43].

Recently, however, several new GPU coherence protocols have been proposed [4, 25, 41] that do not completely invalidate caches at synchronization and thus allow data to be reused across kernels. For example, DeNovo [41] uses ownership with self-invalidating L1 caches to ensure that the cache invalidations only occur for data that are not owned. While any protocol that retains cache data across synchronization points can be exploited by our schedulers, past work has shown that DeNovo strikes a sweet spot between simplicity and efficiency for heterogeneous CPU-GPU systems. Our evaluations therefore assume the DeNovo protocol. However, we evaluate GPU Coherence in Section 7.2 to study the difference between L2-only reuse (GPU Coherence) and L1-and-L2 reuse (DeNovo).

2.2 GPU Scheduling

CUDA [35] and OpenCL [15] support the notion of *thread blocks* or *work groups*, constructs that represent a group of threads that all execute on one GPU core. Without loss of generality, we use *thread block (TB)* in this article. TBs typically have hundreds of threads, making it difficult to execute the entire TB simultaneously. GPUs therefore subdivide TBs in smaller groups of threads called *warps* (CUDA) or *wavefronts* (OpenCL). Without loss of generality, we use *warp* in the rest of this article.

Modern GPUs need three schedulers to schedule threads. A first-level global scheduler assigns kernels to Compute Units (CUs), a second-level global scheduler allocates TBs to CUs, and a third-level per-CU scheduler schedules the warps of those TBs on to the available hardware. Our work concerns TB scheduling to optimize for inter-kernel locality (Section 9 discusses the relationship

with schedulers at other levels). Unlike CPU schedulers, which are part of the operating system software, GPU TB schedulers are implemented in hardware, as expensive CPU-GPU communication in modern heterogeneous systems prohibits management of GPU CUs from a host CPU. The TB schedulers of current GPUs are not well documented, and information about them is scarce [29].

Recent literature has assumed the baseline TB scheduling policy to be a simple *round-robin* (RR) scheme, as TB schedulers of modern GPUs are not publicly available [5, 18, 20, 26, 29, 51]. RR starts by allocating one TB to each CU. It then repeats this process until all CUs are saturated. As CUs start to execute and finish TBs, RR schedules new TBs, one at a time, on demand. Whenever a new kernel is launched, RR starts allocation from the successor of the CU that finished last in the previous kernel. This scheme is simple and cheap. Moreover, it is partially an artifact of simple GPU coherence protocols: Since data are invalidated at kernel boundaries, there is no incentive to implement a scheduler that exploits inter-kernel reuse.

3 THREAD BLOCK DEPENDENCIES

GPU applications that have multiple kernel invocations often have producer-consumer relationships between these invocations—a thread block from one kernel may be a consumer of data that were produced by thread block(s) of previous kernels. In general, the data reuse relationships can be arbitrary. However, we often observe very simple dependencies. The most common type is when a thread block of a kernel consumes the same data it produced in the previous kernel; i.e., thread block i of the current kernel depends on thread block i of the previous kernel. We call such dependencies *self-dependencies*.

We only count producer-consumer dependencies—where data produced by one TB are consumed by another—and immediate dependencies—where the data have to have been produced in the last kernel—as self-dependencies. Accesses to read-only data are not included even though they often exhibit inter-kernel reuse as well. In fact, many times, and especially in irregular applications, read-only accesses constitute a major portion of the total accesses of the TB, and exploiting inter-kernel reuse in them significantly improves performance.

Of course, GPU applications may have other types of dependencies as well, but in an empirical study of the Rodinia, Parboil, and Pannotia benchmark suites, we found that a majority of the dependencies were self-dependencies. We quantified dependencies at a per-TB level as follows: For every unique load performed by a thread, if the data were written by the thread's own TB in the *previous* kernel, a self-dependency was counted. If the data were at the boundary of the TB and was written by a neighboring TB in the *previous* kernel, a ghost zone dependency was counted. If the data were written by any other TB or even written by the same TB but in an *older* kernel, an "other" dependency was counted. The dependency counts were summed for all the loads in all the threads in a TB and for all the kernels in the application to obtain the final (producer-consumer) dependency distribution.

Table 1 shows the dependency breakdown of each TB for the regular applications that we studied (Section 5.4.2).[2] Hotspot, Pathfinder, SRAD v2, and Stencil all predominantly have self-dependencies. SRAD v1 has a significant amount of self-dependencies that are on data produced in older kernels. Therefore, they are not counted as self-dependencies here due to our classification scheme (they are classified as "other"). However, if we were to include these dependencies, SRAD v1 would have 87% self-dependencies. All of Kmeans' producer-consumer dependencies come from the CPU and are thus classified as "other." However, Kmeans does have extremely high self-reuse

[2]Irregular applications have dependencies that are input-dependent and are thus hard to quantify. However, reads of the graph structure (CSR data) do have self-reuse. Note that an *application* can be irregular while its *dependencies across kernels* are not.

Table 1. Producer-consumer Dependence Distribution for Seven Regular Applications (Potential Reuse through Read-only Data Is Not Represented)

Application	Self	Ghost	Other
Hotspot	95%	5%	0%
Kmeans	0%	0%	100%
NW	53%	0%	47%
Pathfinder	100%	0%	0%
SRAD v1	56%	6%	38%
SRAD v2	91%	1%	8%
Stencil	92%	8%	0%

The distributions were computed statically using program information and are over all the kernels of an application. We do not show results for LUD, as it has three kernels, each of which has between 0% and 100% self-dependencies depending upon the particular kernel invocation, and therefore an average dependency distribution does not accurately reflect the true nature of the application. We do not show the distributions for the irregular applications because they are input-dependent and are harder to quantify.

```
main() {
    // Init input and output
    ...

    // Run kernel iteratively
    for (int i = 0; i < N; i++) {
        kernel<<<blocks, 256>>>(input, output);
        swap(input, output);
    }
}
```

```
kernel(input, output) {
    // Global thread ID
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    // Load data
    int me    = input[tid];
    int left  = input[tid - 1];
    int right = input[tid + 1];

    // Compute
    int result = me + (left * right);

    // Store result
    output[tid] = result;
}
```

Fig. 1. Example code template for a generic iterative GPU kernel. The data loads into variables *me*, *left*, and *right* show a self-dependency on the data update performed in the previous kernel.

of read-only data; it is just not captured in Table 1, as it only considers producer-consumer dependencies. NW has only 53% self-dependencies due to its wavefront access pattern. Finally, LUD has three kernels, all of which can have between 0% and 100% self-dependencies based on the specific kernel invocation.

To illustrate why self-dependencies are so prevalent in modern regular applications, we show a generic iterative GPU kernel in Figure 1. This kernel is representative of computation patterns found in most modern iterative GPU applications, such as Hotspot, Pathfinder, SRAD v1, SRAD v2, and Stencil (Section 5.4.2). The CPU repeatedly invokes the GPU kernel for a certain number of iterations. In each kernel invocation, each thread first reads its own value, *me*, and both of its neighbors' values, *left* and *right*. It then performs a simple computation using all three values. Finally, it writes the result to the output array. After the kernel is finished, the input and output arrays are swapped to continue the iterative computation. Due to this swap, each thread reads the value that it wrote in the previous kernel, along with values written by its neighboring threads; i.e., the thread has a self-dependency because of *me*, and the TB as a whole has a self-dependency

because of *me*, *left*, and *right*. Furthermore, it can be seen that threads at the edge of a TB require data from threads in other TBs, which results in a ghost zone dependency.

Thus, by scheduling TB i on the same CU that TB i was scheduled on for the last kernel, a TB scheduler can inexpensively exploit self-dependencies and improve inter-kernel reuse, performance, and energy. For the bulk of this article, we therefore focus on designs optimized for self-dependencies for their simplicity. Section 8 briefly describes how our schedulers can be easily extended to support other types of dependencies. We added this support to our schedulers and found that it did not improve performance and energy any further due to self-dependencies being the predominant dependency in the applications that we studied, justifying our default design choices.

4 IKRA SCHEDULER DESIGN

We describe four hardware TB schedulers that improve on RR (Section 2.2). These schedulers incrementally add features to exploit increasing amounts of data reuse. The first three use a predetermined static allocation of TBs to CUs. The fourth employs work stealing to handle cases where the static schedule results in load-imbalance.

All four schedulers aim to keep hardware cost extremely low. To achieve this goal, we employ *chunking* of TBs, which increases intra-kernel reuse. To isolate this impact, we first describe a chunked version of RR.

4.1 Chunk

The Chunk scheduler (C) is a simple extension of RR. Instead of allocating TBs one-by-one like RR, Chunk groups contiguous TBs into *chunks* and assigns them to each CU at the beginning of the kernel. The size of each chunk is calculated by dividing all the TBs evenly among the CUs. If the number of TBs is not evenly divisible by the number of CUs, the chunks of the first "remainder" number of CUs will have one more TB than the chunks of the other CUs. Thus, each CU is assigned exactly one chunk at the beginning of the kernel.[3] Once a CU has executed its entire chunk, for simplicity, it cannot request more work and must wait for the other CUs to finish. Finally, whenever a new kernel is launched, the assignment of chunks to CUs begins from the successor of the CU that finished last in the previous kernel, similar to RR. This is important for two reasons: First, and most importantly, we wanted to isolate the benefits of intra-kernel locality exploited by our schedulers, and using this scheme allows us to estimate intra-kernel reuse by performing a fair comparison between RR and Chunk. Second, uniform usage of CUs can be beneficial for thermal management and, over the GPU's lifetime, for managing transistor aging evenly across the CUs.

Overhead: The coarse-grained assignment of TBs to CUs results in very low hardware overhead, as each CU only needs to keep track of one chunk. Chunk uses two local registers per CU to store the start and end TB ID of the assigned chunk. The IDs are of type **dim3** and are 63 bits long (x, y, and z coordinates take 31, 16, and 16 bits, respectively, for CUDA). The start and end TB ID are collectively called the *chunk entry* of a CU. Figure 2 shows chunk entries of four different CUs at the launch, middle, and end of a kernel with grid dimensions (6, 4).

An integer divider calculates the chunk size. A comparator compares the start ID with the end ID to determine whether the chunk has been finished. If start ID is less than or equal to end ID, a TB is available. An incrementer increments the start TB ID to denote the launching of a new TB. A global register keeps track of the CU that executed the last TB in the current kernel.

[3] A rare exception is when there are fewer TBs than the number of CUs.

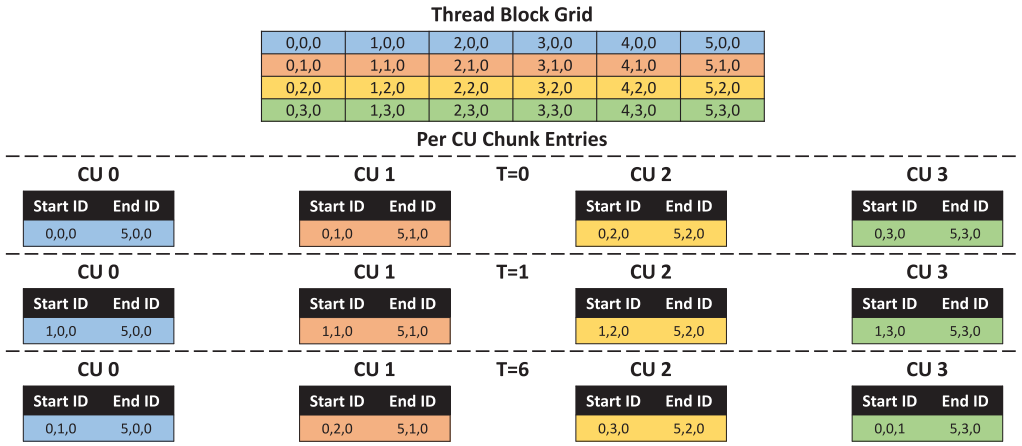

Fig. 2. Example kernel execution with the Chunk scheduler. T=0: Chunk entries of four different CUs when a kernel with a 6x4 TB grid is launched on a system using Chunk. T=1: Chunk entries after each CU has executed one TB. T=6: Chunk entries after each CU has executed its entire chunk. The chunk entries are local to each CU. For clarity, the entries have been color-coded to match the section of the TB grid that they map to.

There is negligible energy and performance overhead as the chunk entry is updated once at the beginning of the kernel, and the start TB ID and global register are only updated when a new TB is launched, all of which are infrequent.

It is important to note that the Chunk scheduler, and all the following schedulers, are *completely decentralized*. Chunk entries are *local* to each CU, and each CU can independently populate them at the beginning of the kernel based on the number of launched thread blocks.

4.2 Reset

The Reset scheduler (R) is identical to Chunk in all aspects except that it always starts allocating chunks from the first CU; i.e., it *resets* to the first CU after every kernel. Although this may result in uneven usage of the CUs over the GPU's lifetime, it allows the scheduler to implicitly handle self-dependencies. As TBs with the same ID are always assigned to the same CU, both the producer and consumer TBs will be scheduled on the same CU in kernels that have self-dependencies, thereby increasing inter-kernel data reuse for these kernels.

Overhead: Reset uses the same hardware as Chunk. However, as chunk assignment always starts from the first CU, the global register to track the last CU is not required.

4.3 Flip

The Flip scheduler (F) builds on Reset by incorporating the following insight: TBs that finished executing last in the previous kernel have the highest likelihood of still having data in the L1 cache. Therefore, consumers of these data should be scheduled first. This can be achieved by simply alternating between starting from the start or end of the assigned chunk. For instance, if CU 0 is assigned TBs 0 through 9, it would execute them from 0 to 9 in the first kernel, 9 to 0 in the second kernel, and so on. Thus, by *flipping* the order of execution each time a kernel is launched, Flip is able to prioritize TBs that have a higher chance of hitting in the L1 cache in the presence of self-dependencies. Since there are multiple TBs running on a CU at any given time and interleaved execution of warps from different TBs complicates scheduling, this technique is only a simple *heuristic* to determine which TBs' data might be in the L1 cache at the end of a kernel.

Overhead: Flip adds an extra direction bit to Reset. The direction bit is used to determine whether to begin execution from the start or end of the CU's assigned chunk. The bit is flipped each time a new kernel is launched (each CU can flip its chunk entry independently). The incrementer is extended to support decrementing, as the end ID needs to be decremented during the flipped execution.

4.4 Steal

All schedulers until this point statically partition work among CUs at the beginning of a kernel. While this approach is simple and exploits self-dependencies for most applications, it sometimes results in a load-imbalanced execution. Different CUs may have different execution times for different TBs due to input dependence, process variation, and network topology. Thus, in the final iteration of our design, we add support for work stealing to find a more balanced allocation of TBs to CUs, while still maintaining as much inter-kernel locality as possible. This is particularly important for irregular workloads, as they are becoming increasingly popular on GPUs [6]. We call the resulting scheduler Steal (S).

Whenever a CU X has executed all the TBs in its chunk entry, it attempts to *steal* one TB from its left neighbor. If the left neighbor does not have any available TBs, the right neighbor is checked. If the right neighbor does not have any available TBs either, the stealing CU continues sequentially until it finds a CU Y that has an available TB. If no available TBs are found, it implies the kernel has finished execution. We opt for this stealing policy, as it is simple and has low overhead. The CUs communicate via simple custom hardware messages.

To support locality-preserving work stealing, S implements a per-CU data structure called the **steal queue**. The steal queue of a CU keeps track of stolen TBs. This allows us to exploit locality for these TBs by executing them on the same CU in the subsequent kernels. To steal a TB from a chunk entry, CU X first checks CU $Y's$ direction bit, as it dictates which TB is stolen. If execution began from start ID, the "end" TB is stolen and CU $Y's$ end ID is decremented; if execution began from end ID, the "start" TB is stolen and CU $Y's$ start ID is incremented. CU X then inserts the stolen TB in its steal queue and executes it. Stealing continues for as long as the kernel is not finished. Figure 3 shows an example in which CU 0 finishes executing its chunk and steals TB (5,1,0) from CU 1. CU 0 decrements CU 1's chunk's end ID and inserts the stolen TB in its own steal queue.

Once the current kernel has finished execution and a new kernel is launched, each CU should execute the same TBs it executed in the previous kernel to exploit inter-kernel locality based on self-dependencies. These TBs consist of both the CU's chunk entry and the steal queue from the last kernel invocation.[4] To support this, both the chunk entry and steal queue must be preserved. As a result, the start and end IDs of the chunk entry can no longer be modified to keep track of the currently executing TB (Section 4.1). Instead, the ID of the currently executing TB is kept in a separate register, which is compared against the bounds of the chunk entry to determine whether the entire chunk has been executed. The steal queue uses standard head and tail pointers to determine the next TB to execute and whether all the steal queue TBs have been executed.

To further maximize inter-kernel locality, S extends the concept of flip (Section 4.3) to choose the order of execution between the chunk entry and steal queue. If the chunk entry is executed first and steal queue second in kernel k, then in kernel $k + 1$, the steal queue is executed first and the chunk entry second. Furthermore, the TBs inside the steal queue are flipped as well to ensure younger TBs are executed first.

[4]The steal queues may not stabilize if consecutive kernels have a different load balance. In such a case, S will achieve load balance but will lose locality.

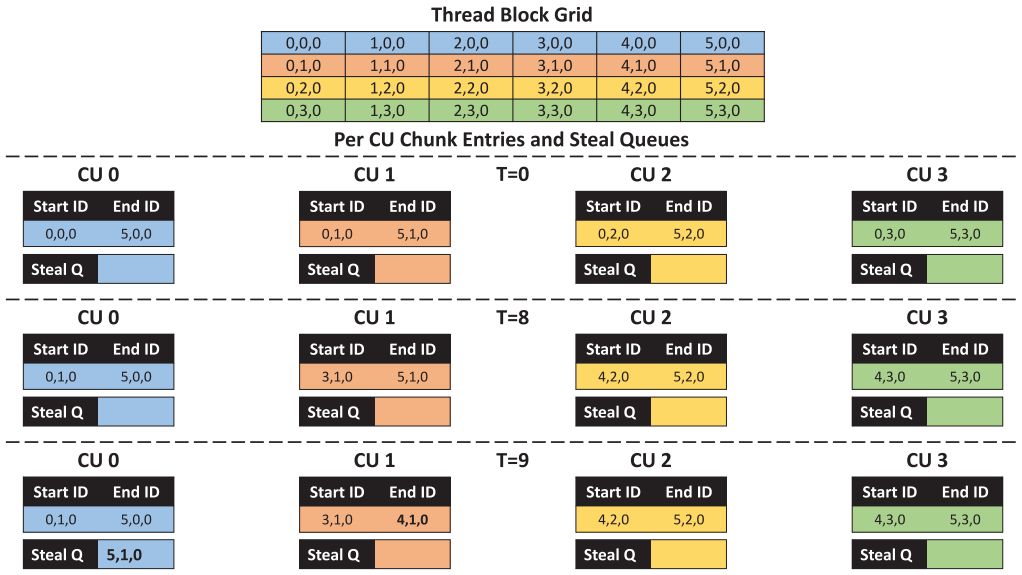

Fig. 3. Example kernel execution with the Steal scheduler. T=0: Chunk entries of four different CUs when a kernel with a 6x4 TB grid is launched on a system using Steal. T=8: CU 0 finishes executing its entire chunk and attempts to steal a TB from CU 1. T=9: CU 0 steals TB (5,1,0) from CU 1 and (1) inserts it in its steal queue and (2) updates CU 1's chunk entry by decrementing the end ID. The chunk entries are local to each CU. For clarity, the entries have been color-coded to match the section of the TB grid that they map to.

In our experiments, the steal queues are sized at 32 entries each, so it is possible that they may overflow due to too much stealing. In this case, we cannot keep track of the evicted TBs. To address this, whenever an overflow occurs, a broadcast is sent to all other CUs notifying them that an overflow has occurred and the scheduler must be reinitialized once the current kernel finishes. When the next kernel launches, the chunk entries and steal queues are discarded, and chunk assignments are performed again. Alternatively, stealing can be disabled when the steal queue becomes full. Doing so would preserve more locality at the cost of load-imbalance. We chose not to do so to prioritize load-balance, as that is more important for irregular applications (Section 6.3). To keep the size of steal queues in check, we attempt to steal from them before stealing from a CU's chunk entry. In our experiments, overflows were extremely rare and typically did not occur more than once during the entire execution of the application. The scheduler is also reinitialized when a new kernel with different dimensions is launched.

The overall algorithm is shown in Algorithm 1. For brevity, the algorithm is shown without the direction bit. Functions *StealFromQ* and *StealFromChunk* (lines 35 and 39, respectively) show the actions that need to be performed during a steal operation. Function *StealTB* (line 17) shows the overall stealing algorithm described above—checking steal queues (lines 18–25) before chunk entries (lines 26–33), and checking left neighbors (lines 18 and 26) and right neighbors (lines 20 and 28) before sequentially checking the remaining CUs (lines 23 and 31).

The overall function *S* for each CU has three parts: initialization (lines 2–9), steal queue and chunk entry execution (lines 10–11), and stealing (lines 12–16). During initialization, there are two possible scenarios. If there were no previous kernels or the *reinit* flag was set due to steal queue overflow or the kernel's dimension are different from the previous kernel, the scheduler is reinitialized: TBs are chunked again, the "current TB" register is reset, and the steal queue is cleared. Otherwise, both the "current TB" register and steal queue pointers are reset to begin

ALGORITHM 1: Pseudocode for the work-stealing scheduler.

```
1: function S
2:     if firstKernel or reinit or dimChange then
3:         ChunkTBs( )
4:         ResetCurrentTBRegister( )
5:         ClearStealQ( )
6:     else
7:         ResetCurrentTBRegister( )
8:         ResetStealQPointers( )
9:     reinit ← false
10:    ExecuteStealQ( )
11:    ExecuteChunk( )
12:    while !done do
13:        tb ← StealTB()
14:        Execute(tb)
15:        stealQ.insert(tb)
16:        if stealQ.size = 32 then reinit ← true
17: function StealTB
18:    if left.stealQ.available then
19:        return StealFromQ(left)
20:    else if right.stealQ.available then
21:        return StealFromQ(right)
22:    else
23:        for all CUs do
24:            if cu.stealQ.available then
25:                return StealFromQ(cu)
26:    if left.chunk.available then
27:        return StealFromChunk(left)
28:    else if right.chunk.available then
29:        return StealFromChunk(right)
30:    else
31:        for all CUs do
32:            if cu.chunk.available then
33:                return StealFromChunk(cu)
34:    done ← true
35: function StealFromQ(cu)
36:    tb ← cu.stealQ.tail
37:    cu.stealQ.tail ← cu.stealQ.tail − 1
38:    return tb
39: function StealFromChunk(cu)
40:    tb ← cu.chunk.end
41:    cu.chunk.end ← cu.chunk.end − 1
42:    return tb
```

executing the appropriate TB. After initialization, the CU executes both its steal queue and chunk entry. Once a CU's steal queue and chunk entry have both been executed, the CU starts stealing TBs from other CUs until all TBs have been executed and the kernel is finished. If at any time the CU's steal queue becomes full, it sets the *reinit* flag, which notifies all the other CUs to reinitialize the chunk entries in the next kernel.

Overhead: Steal is the only hardware scheduler that requires additional SRAM. The steal queues hold 32 **dim3** entries each. Consequently, the total SRAM for storing steal queues is 2,016 bits or 252 B per CU. An additional register is required to keep track of the currently executing TB's ID, and an additional direction bit is needed to choose the order of execution between the chunk entry and steal queue. The other area overheads are the same as Flip, which are negligible. In our simulated system (Section 5), the combined storage of the register file, scratchpad, L1 cache, and local L2 bank of each CU is 528 KB, which makes the SRAM area overhead of Steal less than **0.05%**. Moreover, the area requirements of the scheduler are independent of CU size, resulting in even lower relative overhead as cache, scratchpad, and register file sizes increase. Execution time and energy overhead are also small due to the fact that the stealing logic is used at most once per TB (in the worst case), which is infrequent compared to the other computations performed by the CU.

4.5 Summary

The IKRA schedulers build on top of one another to address the shortcomings of RR by exploiting the behavior of modern GPU applications. The most salient aspects of these schedulers are that they are simple, decentralized, and have extremely low overhead. They also provide a range of design points to cater to both current and emerging applications. C, R, and F use simple optimizations to maximize inter-kernel locality in current, regular applications. S adds locality-preserving work stealing to maximize performance in emerging, irregular applications.

Table 2. Default Simulated System Parameters for Both Microbenchmarks and Applications

Core Parameters	CPU	GPU
Cores	1	3, 15
Frequency	2 GHz	700 MHz
L1 Size	**L1 Hit Latency**	**Remote L1 Hit Latency**
32 KB, 128 KB (8-way assoc.)	1 cycle	35–83 cycles
L2 Size	**L2 Hit Latency**	**Memory Latency**
256 KB, 4 MB (16 banks, NUCA)	29–61 cycles	197–261 cycles
MSHRs	**Store Buffer Size**	
64 entries	64 entries	

The smaller parameter for CUs, L1 size, and L2 size corresponds to the microbenchmark configuration. The microbenchmarks use a smaller configuration for simplicity.

5 METHODOLOGY

5.1 Baseline System Architecture

Although our work only targets the GPU, we model a tightly integrated CPU-GPU system with a unified and coherent memory address space to reflect an emerging platform. We expect our schedulers to benefit and our qualitative insights to hold for discrete GPU systems as well.

The CPU cores and GPU CUs are connected via a mesh interconnection network. Each network node contains either a CPU core or a GPU CU with an L1 cache (local to the CPU core or GPU CU) and a bank of the shared L2 cache (shared by all CPU cores and GPU CUs). All L1 caches are coherent and use a writeback policy.

Although any GPU coherence protocol that allows data reuse across kernels can be used [4, 41], we chose the DeNovo coherence protocol because of its simplicity and lower overhead. Although we focus on DeNovo, we also report results for GPU Coherence in Section 7.2. Since our baseline architecture is similar to prior work that extended DeNovo to heterogeneous systems [41], we obtained the simulator from the authors and extended it to use our new TB schedulers.[5] The next section describes the simulation environment and specific system parameters modeled.

5.2 Simulation Environment and Parameters

Table 2 summarizes the key parameters of the system, which is the same as prior work [41] except for its L1 cache size. Our cache sizes are similar to the NVIDIA Volta architecture, which has 128 KB L1s and a 6 MB L2 [36]. Our overall system is similar to modern mobile SoCs, such as Samsung Exynos 9820 [53]. The simulator uses the Simics full-system functional mode to model the CPUs [50], the Wisconsin GEMS memory timing simulator [34], and GPGPU-Sim v3.2.1 [5] to model the GPU.[6] The simulator uses Garnet [2] to model a 4×4 mesh interconnect with a CPU core or a GPU CU at each node. To measure energy consumption, the GPU CUs use GPUWattch [28] and the NoC energy measurements use McPAT v.1.1 [33]. We do not measure idle energy under the assumption that the CUs implement clock- and power-gating and can be turned off when idling. We also do not model the CPU core or CPU L1 energy, as our design is implemented purely on the GPU.

[5] Another reason we used an integrated CPU-GPU system was because the original authors of DeNovo had implemented the protocol in such a system.

[6] GPGPU-Sim is the only CUDA-capable open-source cycle-accurate GPU simulator, has been validated against real hardware [5], and has been used in many academic studies [1, 8, 9, 18–20, 22, 26].

The number of concurrent warps, TBs per CU, and the warp scheduler can affect how much useful data remain in the L1 cache after a kernel finishes. Although fine-tuning these parameters can result in potentially higher inter-kernel data reuse (Section 8), we use GPGPU-Sim's default GTO warp scheduler, along with the default concurrency limit of 48 warps and 8 TBs per CU.

Although recent GPUs support multiple kernels executing concurrently, such kernel-level schedulers are not the focus of our work. Without loss of generality, we assume a single kernel executing at a time in our experiments. Section 9 discusses the interaction between kernel-level schedulers and our TB schedulers.

Most of our applications only use the GPU for computation. Two applications, Kmeans and SRAD v2, use the CPU for computation. They do not, however, parallelize the CPU portion, which dwarfs the GPU execution and artificially skews the overall execution time. We, therefore, only report the GPU execution time for all our applications.

We do not explicitly model the performance and energy overhead of our schedulers, because they are expected to be negligible—they are invoked infrequently (at most once per TB), the hardware is small, and most accesses are off the critical path (Section 4).

5.3 Configurations

To evaluate our schedulers, we simulate the following configurations: G = (GPGPU-Sim) round-robin; C = Chunk; R = Reset; F = Flip; and S = Steal. All configurations have the same system parameters (Section 5.2) and differ only in the choice of scheduling policy used. Furthermore, all configurations use DeNovo, ensuring that benefits only come from IKRA scheduling and not from the coherence protocol. In Section 7.2, all configurations use GPU Coherence.

5.4 Workloads

To showcase the potential benefits that IKRA TB scheduling can achieve, we first examine four simple microbenchmarks that are designed to exploit inter-kernel reuse. The microbenchmarks are designed to understand fundamental issues in workloads with a high degree of data reuse. We then evaluate our schedulers on several benchmarks from CUDA SDK [35], Rodinia [7], Parboil [44], and Pannotia [6] suites. These benchmarks illustrate varying degrees of data reuse in more complex kernels.

5.4.1 Microbenchmarks. We evaluate four microbenchmarks: *GPU-small*, *GPU-medium*, *GPU-large*, and *Work-Stealing*. We use one CPU core and three GPU CUs for all microbenchmarks. In each microbenchmark, the CPU invokes a GPU kernel 16 times. Each thread of this kernel does load-add-store for a number of memory locations. The number of load-add-store operations is input-based in Work-Stealing (described below) and 8 in the remaining microbenchmarks. This is done to obtain a reasonable amount of work per thread.

All microbenchmarks only perform computation on the GPU. The difference between the three GPU-* microbenchmarks is their input size: all data fitting in the L1 (GPU-small), data split between the L1 and L2 (GPU-medium), and data mostly split between the L2 and memory (GPU-large). The purpose of these different input sizes is to explore the behavior of the schedulers as input size increases. The input size only affects the number of TBs in the system.

Work-Stealing demonstrates the capability of the scheduler to maximize data reuse while achieving load balance. This microbenchmark consists of a kernel in which each TB performs between 1 and 64 load-add-stores based on the input. The input size is the same as GPU-large.

5.4.2 Applications. We examine 17 popular regular and irregular applications from CUDA SDK [35], Rodinia [7], Parboil [44], and Pannotia [6], listed in Table 3 with their input sizes, TB sizes, number of TBs, and the total data accessed per kernel. For the regular applications, we used

Table 3. Applications with Their Input Size, TB Size, #TBs, and the Amount of Data Accessed in Each Kernel

Application	Input Size	TB Size	#TBs	Total Data Accessed Per Kernel (KB)
Regular iterative applications				
Hotspot (HS) [7]	256 × 256, 20 iters	16 × 16	361	768
Kmeans (KM) [7]	25 K × 35	64	391	3,516
LU Decomposition (LUD) [7]	768 × 768	16, 32, 16 × 16	1–2,209	1–2,303
Needleman-Wunsch (NW) [7]	512 × 512	8	1–63	0.5–31.5
Pathfinder (PF) [7]	10 × 100 K, 10 iters	256	394	1,172
SRAD v1 (SD1) [7]	256 × 256, 10 iters	512	128	256/768/512/2,560/2,048/256
SRAD v2 (SD2) [7]	512 × 512, 10 iters	16 × 16	1,024	6,144
Stencil (ST) [44]	128 × 128 × 8, 10 iters	32 × 4	64	1,024
Regular non-iterative applications				
Backprop (BP) [7]	32 K	16 × 16	2,048	4,676
ConvolutionSeparable (CP) [35]	1024 × 512	16 × 8, 16 × 4	512, 1,024	4,096
LavaMD (LM) [7]	8	128	8	61
Nearest Neighbor (NN) [7]	171,056	256	669	2,005
SGEMM (SG) [44]	A: 1024 × 992, B: 992 × 1056	16 × 8	496	12,284
Irregular iterative applications				
Dijkstra (DJ) [6]	(1) cond_mat: 633 KB	256	66	–
	(2) olesnik0: 4,525 KB		345	–
Graph Coloring (CL) [6]	(1) cond_mat: 699 KB	256	66	–
	(2) olesnik0: 4,869 KB		345	–
MIS (MI) [6]	(1) cond_mat: 578 KB	256	66	–
	(2) olesnik0: 3,641 KB		345	–
PageRank (PR) [6]	(1) cond_mat: 382 KB	256	66	131/382
	(2) olesnik0: 2,607 KB		345	690/2,607

Multiple entries for TB size, #TBs, and data accessed per kernel are for different kernels or kernel invocations in the application. The data accessed in each kernel of DJ, CL, and MIS are not reported, because they are dependent on the frontier of the graph and cannot be computed statically (the input size column provides a loose upper bound). All four irregular applications are run with two graphs: (1) cond_mat [45] and (2) olesnik0 [46]. SRAD v1 performs all computation on the GPU while SRAD v2 splits computation between the CPU and GPU and uses the scratchpad in the GPU kernel.

inputs similar to previous work on GPU cache reuse [47, 55]. For the irregular, graph-based applications (Color, Dijkstra, MIS, and PageRank), we studied several graphs from Matrix Market graphs [10] and show results for two representative graphs—one balanced (olesnik0) and one imbalanced (cond_mat).

We differentiate between regular and irregular applications, as they present different trade-offs for our designs. Furthermore, although our technique targets iterative applications with inter-kernel reuse, we also include results for non-iterative applications to show the robustness of our designs. We briefly discuss results for non-iterative applications in Section 6.4.

We modified the applications to work in a unified memory address space. Additionally, for Stencil, we added padding to the input and output arrays to alleviate anomalous conflict misses with our input size. The storage overhead of this modification was 0.5%.

Finally, we used the default TB sizes specified by the benchmark suites in all of our applications (Table 3). We do not expect our qualitative conclusions to be affected by other TB sizes that would be used in practice.

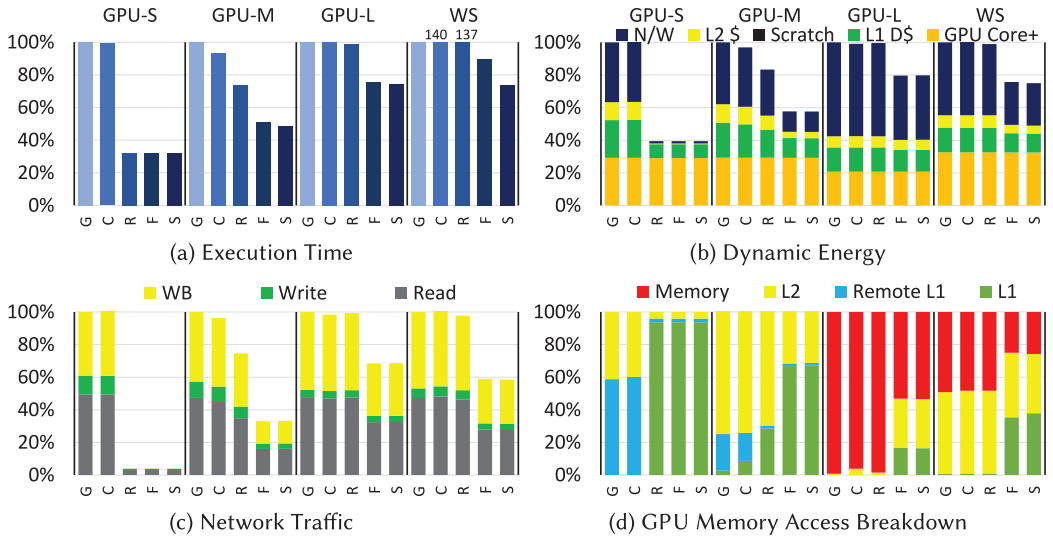

Fig. 4. Comparison of (a) execution time, (b) dynamic energy, (c) network traffic, and (d) GPU memory access breakdowns of all schedulers for four different microbenchmarks. All bars are normalized to G.

6 RESULTS WITH BASELINE CONFIGURATION

6.1 Microbenchmarks

Figure 4 shows the execution time, energy, network traffic, and GPU memory access breakdown for our microbenchmarks for various configurations. The energy bars are subdivided by where the energy is consumed: GPU core+,[7] L1 cache, scratchpad, L2 cache, and network. Network traffic bars are subdivided by message type: read, write, or writeback. Memory bars are subdivided by where the memory accesses were served from: L1, remote L1, L2, and main memory.

The first microbenchmark, GPU-small, illustrates the benefits of scheduling producers and consumers of data on the same CU. The total working set of this microbenchmark is such that a CU's L1 cache exactly holds the combined working set of all the TBs it executes. Therefore, it does not matter which order the TBs are executed in, as long as they are executed on the correct CU. As a result, a scheduler as simple as R improves significantly over G, which is oblivious to the dependencies that exist between TBs and has to go to remote L1s and the L2 to service memory requests. Once the cache is warmed up, R has an L1 cache hit rate of 100% and correspondingly *no* network traffic, reducing execution time and energy by 68% and 61%, respectively. Since the working set fits in the L1 cache and the workload is balanced, F and S perform the same as R.

GPU-medium demonstrates the importance of prioritizing TBs when the working set is split between the L1 and L2. In such a scenario, threads whose data might still be in the cache should be prioritized over threads whose data have likely been evicted. Since the working set is now bigger than the L1 cache of a CU, by the time a kernel finishes, the data from the older TBs have been evicted by data from the younger ones. Consequently, R does not perform as well as it did in GPU-small. However, it still sees a moderate improvement, as not all useful data are evicted from the L1. However, F schedules TBs that require data from the younger previous TBs first, decreasing both execution time and energy by 31% compared to R. This prioritization is evident from the doubling of the L1 hit rate.

[7]GPU core+ includes the instruction cache, constant cache, register file, SFU, FPU, warp scheduler, and the core pipeline.

Increasing the working set size such that it exceeds the aggregate capacity of all the L1s and the L2 further emphasizes the importance of TB execution order. As Figure 4(d) shows, for GPU-large, R is unable to exploit any locality whatsoever and all its memory requests have to be serviced by main memory (same as G). F approximately halves the number of main memory accesses and increases the L1 and L2 hit rates to 17% and 29%, respectively. Compared to R, this decreases execution time by 23% and energy by 20%.

Finally, Work-Stealing shows the benefits of locality-preserving work stealing (S) when an application is inherently irregular and a static allocation of TBs to CUs results in a load-imbalanced execution. The Work-Stealing microbenchmark performs a different amount of work in each TB based on the input to each thread in the TB. Consequently, the simple block allocation schemes of C, R, and F suffer from load-imbalance. C and R both observe a slowdown, as they neither exploit locality nor provide load balance. F is able to slightly improve performance due to the improved locality from TB prioritization. S, however, is able to decrease execution time by 26% compared to G, as it allows CUs to steal TBs and achieve a better load-balance without sacrificing locality, as indicated by Figure 4(d). Note that F and S consume the same amount of energy because, as discussed in Section 5.2, we do not report the reduction in idle time energy.

These results show that there is significant potential for performance and energy improvement by using simple, yet effective, TB schedulers that do not cost much more than G.

6.2 Regular Applications

Figure 5 shows the execution time, energy, network traffic, and GPU memory access breakdown for our eight regular, iterative applications. Compared to G, S is the fastest scheduler, reducing execution time by 19% (max 43%), energy by 11% (max 21%), and network traffic by 42% (max 86%) on average. Compared to C, which only provides intra-kernel locality, S reduces average execution time, energy, and network traffic by 16%, 8%, and 32%, respectively, elucidating the improvements from exploiting inter-kernel locality. Furthermore, F and S have approximately the same benefits on average, as the static schedule created by F works well for these structured, regular applications, and work stealing is not required to create a better schedule. We next explain these results in further detail, starting from understanding the impact on cache hit rate.

6.2.1 Cache Hit Rate. As shown in Figure 5(d), all four IKRA schedulers increase locality at the L1 cache compared to G. C has the smallest increase in L1 hit rate, as it only increases intra-kernel reuse. The other three schedulers have higher hit rates, as they also exploit inter-kernel reuse. The highest locality is achieved by F, although it is only slightly better than R and S—R is worse because it does not prioritize TBs, while S is worse because stealing results in a loss of L1 locality. Hotspot, NW, and Stencil are three examples of applications in which G had an L1 hit rate of close to zero, and IKRA scheduling significantly increased the L1 hit rate (to 91%, 30%, and 56%, respectively) in the best case (F). Overall, on average, S *doubles* the L1 hit rate and reduces trips to main memory by 31%.

6.2.2 Network Traffic. The changes in GPU cache hit rate are directly translated into changes in network traffic, as shown in Figure 5(c). The main source of improvement is the reduction in read and write traffic due to increased cache hit rate. Hotspot, SRAD v1, and Stencil see the biggest improvement, because they have both extremely high intra-kernel and inter-kernel locality. S has slightly higher network traffic than F, as stealing causes a loss of locality during the kernel in which it occurs. Overall, S reduces network traffic by 42% on average (max 86%).

6.2.3 Energy. Figure 5(b) shows that on average 58% of a CU's energy consumption comes from the core; i.e., compute. Therefore, the decrease in energy is not of the same magnitude as

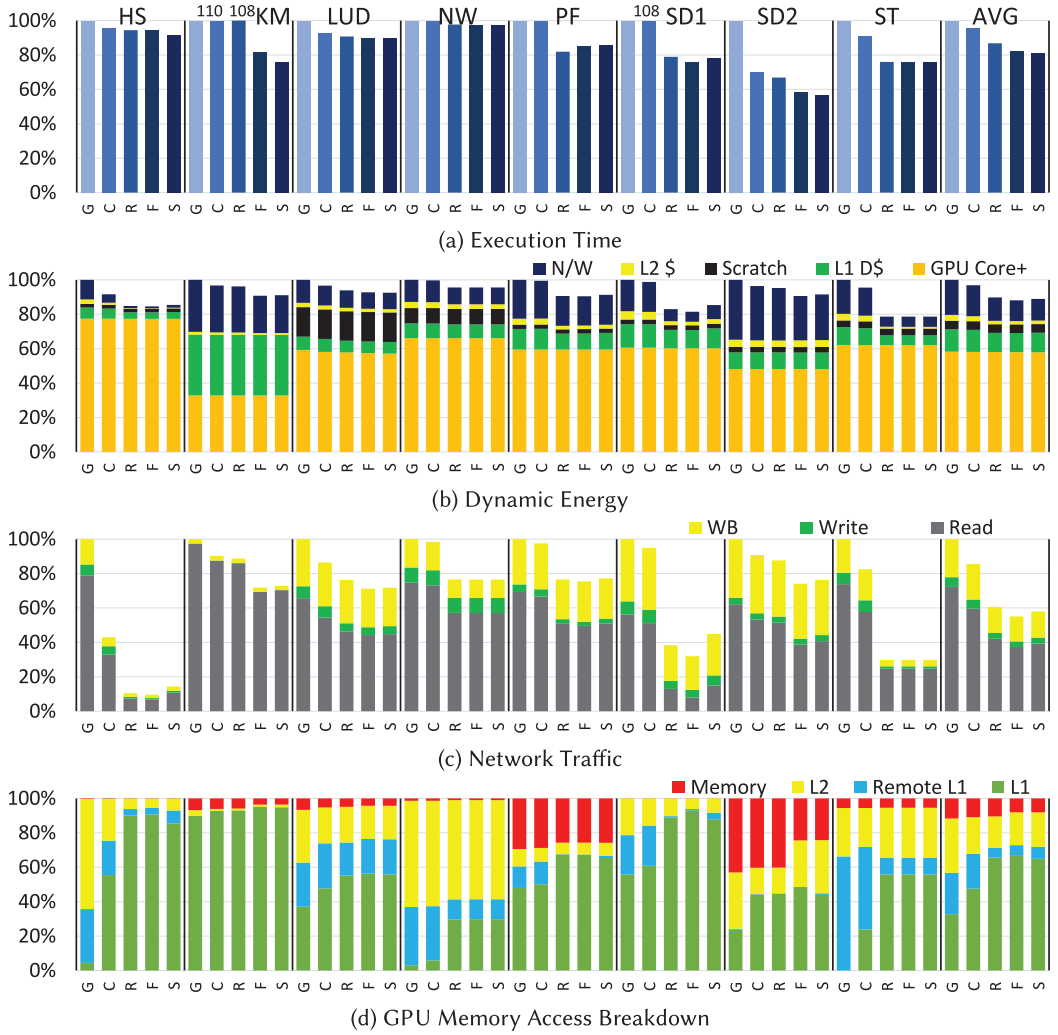

Fig. 5. Comparison of (a) execution time, (b) dynamic energy, (c) network traffic, and (d) GPU memory access breakdowns for the four schedulers for regular applications with multiple kernel invocations. All bars are normalized to G.

the decrease in network traffic, although it is proportional. Moreover, applications such as LUD and NW perform the majority of their memory accesses in the scratchpad, further decreasing the impact of improved L1 hit rates. Overall, R, F, and S have comparable energy for these applications, on average consuming 10%, 12%, and 11% less energy than G (max 21%), respectively. S also decreases idle time energy, but we do not include it, as discussed in Section 5.2.

6.2.4 Execution Time. Figure 5(a) shows that on average all schedulers are able to reduce execution time compared to G. Kmeans, SRAD v1, SRAD v2, and Stencil show the biggest improvement in execution time (24%, 22%, 43%, 24%, respectively), as they have the highest reduction in costly main memory accesses (Kmeans and SRAD v2) and the highest difference in L1 hit rate from G (SRAD v1 and Stencil). NW sees relatively little improvement because it is compute- and scratchpad-intensive, and the amount of reuse between producers and consumers is low. Furthermore, there

are only three instances in which any IKRA scheduler does worse than G (C and R in Kmeans, and C in SRAD v1) and even in these cases, the best IKRA schedulers are always better than G.

In both applications where IKRA does worse than G for some scheduler, C increases locality but at the expense of creating an extremely load-imbalanced schedule due to the data-dependent nature of the kernels. This results in a slower execution time. In Kmeans, R behaves the same as C, as it is unable to increase inter-kernel reuse. However, in SRAD v1, R improves inter-kernel locality significantly enough to improve overall performance. F improves performance in both applications, whereas S helps Kmeans due to locality-preserving load-balancing but slightly hurts SRAD v1 due to stealing, resulting in lost opportunities for reuse. In both cases, S significantly outperforms G.

Overall, for these eight applications, our schedulers improve cache hit rate, network traffic, energy, and performance compared to G in all but three cases. The best-performing scheduler is *always* better than G. With the exception of Kmeans, S is only marginally beneficial, as these applications are generally load-balanced. Thus, considering the slightly higher overhead of S, F achieves the sweet spot in terms of performance, energy, and overhead for these eight regular applications.

6.3 Irregular Applications

Figure 6 shows the execution time, energy, network traffic, and GPU memory access breakdown for our four irregular, iterative applications. It is immediately apparent that C causes a significant slowdown in six out of the eight experiments—the static chunking and allocation of TBs to CUs is no longer load-balanced due to the input-dependent nature of the kernels and results in a slower execution, as only a few CUs perform most of the work. Due to this reason, S is indispensable for performance. It is, again, the fastest scheduler and consumes only slightly (2%) more energy than F. Compared to G, S reduces average execution time by 10% (max 18%), energy by 8% (max 19%), and network traffic by 16% (max 37%). We next discuss these results in further detail, starting by analyzing the impact on cache hit rate.

6.3.1 Cache Hit Rate. As shown in Figure 6(d), most of the irregular applications we study have high L1 hit rates to begin with, as dense reads of the read-only graph data exhibit excellent spatial locality. Furthermore, the working set of synchronization variables (shared graph data) is sufficiently small that it fits in the L1 caches of the CUs, resulting in far fewer accesses to the L2 and main memory than in regular applications. Compared to regular applications, fewer dependencies in these irregular applications are self-dependencies, as the dependencies are mostly dictated by the structure of the graph. Nonetheless, R, F, and S are still able to take advantage of inter-kernel dependencies due to the connectivity of the input graphs and the chunking of neighboring TBs on the same CU. For instance, in DJ2, the input graph is connected in such a way that a TB reuses its neighboring TB's (which resides on the same CU) data from the previous kernel. Chunking also has a counter-intuitive impact on load-balance in certain cases: Even though DJ2 has a more balanced input graph than DJ1, chunking makes DJ2 more load-imbalanced in practice.

6.3.2 Network Traffic. As shown in Figure 6(c), network traffic follows the same trends as the cache hit rates. Graph Coloring sees the least improvement, as it does not have any self-dependencies, and inter-kernel reuse is only exploited through dependencies based on graph connectivity. Dijkstra sees the most improvement, as it has the most self-dependencies. It is interesting to note that S consistently has higher network traffic than F. This is because the inner loop of all four irregular applications has two or more static kernels, with each kernel possessing a different load-balance. As a result, S never finds one stable schedule and keeps stealing in each kernel, resulting in higher network traffic. Overall, S only reduces network traffic by 16% on average (max 37%).

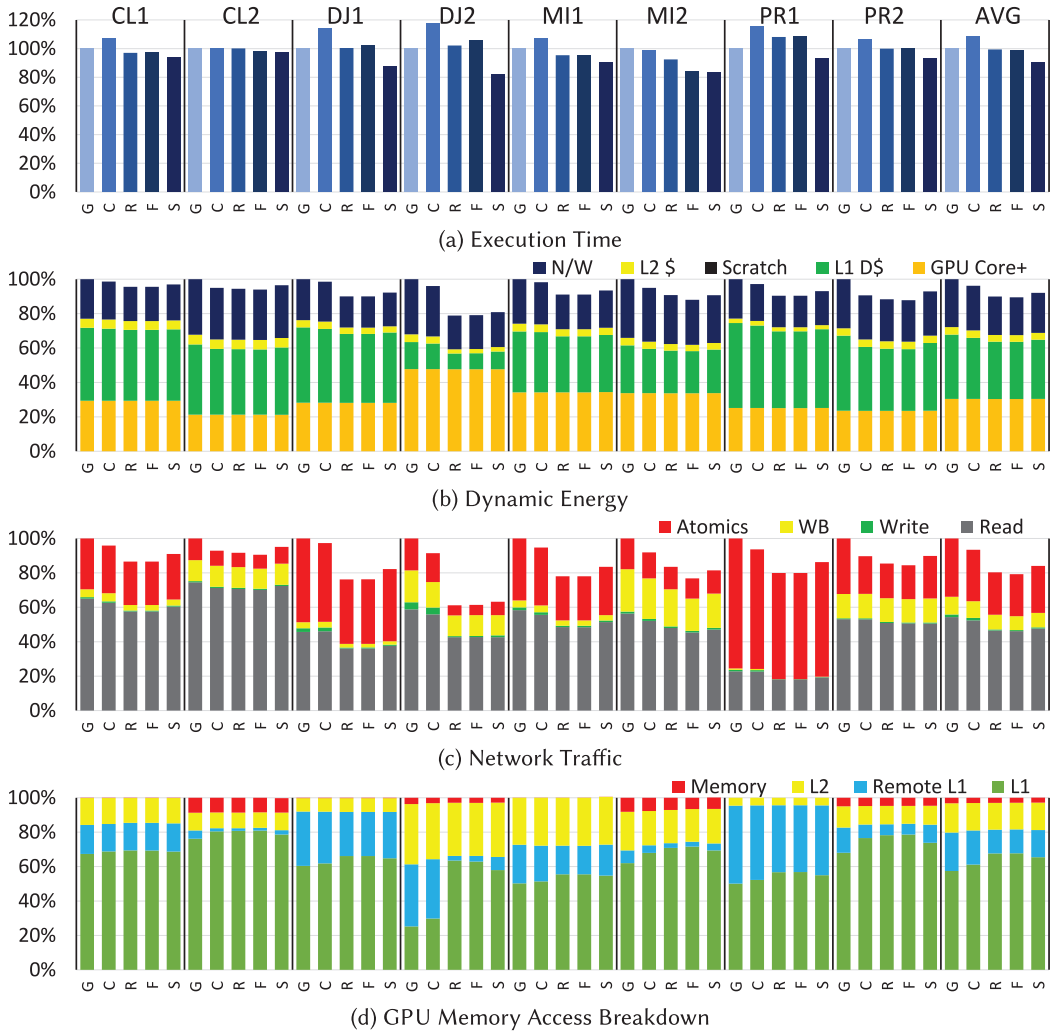

Fig. 6. Comparison of (a) execution time, (b) dynamic energy, (c) network traffic, and (d) GPU memory access breakdowns for the four schedulers for irregular applications with two inputs each (normalized to G). Part (d) contains both data and synchronization accesses.

6.3.3 Energy. As Figure 6(b) shows, on average, 70% of a CU's energy consumption comes from the memory system, with 32% coming from the L2 and network. Therefore, there is significant room for energy improvement. However, the actual improvement is rather modest for two reasons. First, as described in Section 6.3.1, very few dependencies in these applications are self-dependencies, limiting the amount of inter-kernel reuse. Second, as described in Section 5.2, we do not report idle time energy in our experiments. Consequently, even though S is able to significantly decrease idle time, it is not reflected in its energy savings. Overall, CL1 and CL2 have the smallest reduction in energy, while DJ2 has the most due to a locality-friendly input graph structure. On average, S consumes 8% less energy than G, but consumes 3% more than F due to its higher network traffic (Section 6.3.2) and missing idle time energy savings.

6.3.4 Execution Time. Finally, Figure 6(a) shows that locality-preserving work stealing noticeably improves performance in irregular applications. Even though F is able to reasonably increase inter-kernel data reuse, its static allocation of TBs to CUs is load-imbalanced due to the inherent nature of these applications. In several cases (DJ1, DJ2, PR1), F does even worse than G. However, S is *always* able to create a more balanced schedule while maintaining inter-kernel locality at the same time. Whereas F only improves average execution time by 1% compared to G, S improves it by 10%.

An interesting result is that F adversely affects Dijkstra with either input. This is because F causes the vertex frontier to reside predominantly in the L1 caches of the CUs, as opposed to the L2 cache for R. Thus, when TBs access vertex frontier data, they have to fetch it from remote L1s, which is more expensive than accessing L2 banks, negatively affecting performance.

Overall, these results show that focusing solely on increasing inter-kernel reuse is insufficient; locality-preserving work stealing is necessary to harmonize locality and load-balance.

6.4 Non-iterative Applications

Our five non-iterative (regular) applications have *no* inter-kernel reuse (since they do not have multiple kernels). We briefly summarize results for them with the IKRA schedulers only to ensure there is no negative impact. Nevertheless, since the schedulers do exploit some intra-kernel reuse due to chunking, we saw some positive impact from C. NN also showed positive impact from S due to better load balance (since these are regular applications, the benefit is small). Overall, S reduces the execution time over G by 5% on average (max 16% for Backprop), most of which comes from C. No scheduler performs worse than G. The impact on energy is negligible.

We also ran our eight irregular workloads with a single iteration to model non-iterative irregular applications. Chunking degraded performance but S was able to recover it via load-balancing. Overall, S reduces execution time over G by 4% on average (max 12% for MI1) as a result of exploiting intra-kernel locality (due to chunking) while maintaining load-balance. Compared to G, CL1 and PR1 are 3% and 5% slower, respectively, with S as the scheduler is unable to achieve load-balance within a single iteration. However, as shown in Section 6.3, both CL1 and PR1 are faster with S than G once inter-kernel reuse gets exploited and load-balance is achieved over several iterations.

7 SENSITIVITY ANALYSIS

7.1 Cache Size Scaling

To understand how our IKRA schedulers perform with various cache sizes, we performed experiments with L1 cache sizes from 16 KB to 512 KB and corresponding L2 cache sizes from 512 KB to 16 MB. The results of these experiments can also be used to understand the behavior of the IKRA schedulers with a fixed cache size and varying input size. We performed these experiments for all our applications but show results for only four for brevity. The remaining applications follow similar trends.

Figure 7 shows the normalized execution time of LUD, SRAD v1, SRAD v2, and DJ2 with the various schedulers and L1 cache sizes ranging from 16 KB to 512 KB. An application's results are normalized to its execution time with G with a 128 KB L1 cache, our default configuration.

We make the following four observations from Figure 7. First, in general, the performance trend is as expected—the larger the L1 cache, the better the IKRA schedulers perform over G; however, the improvement plateaus once the cache is large enough to hold the entire working set size of the application; e.g., in LUD and SRAD v1.

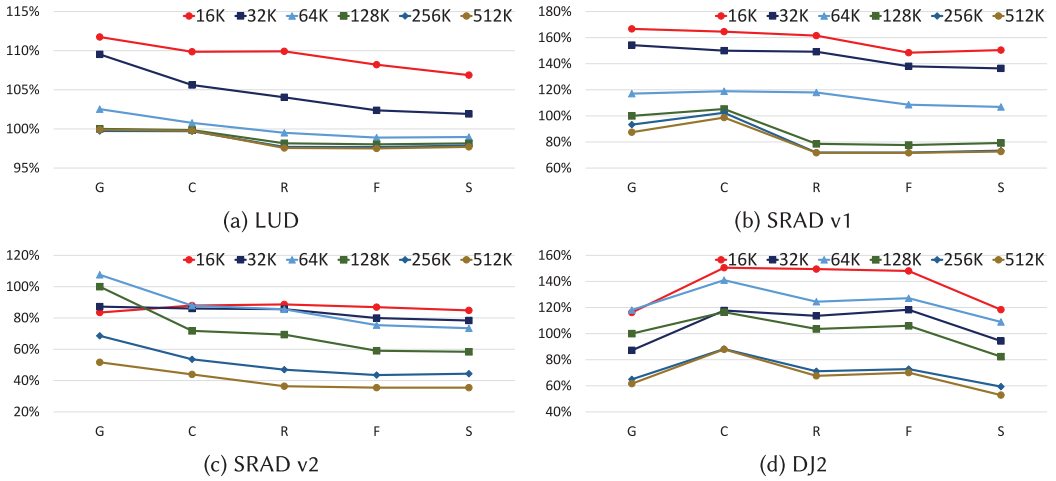

Fig. 7. Normalized execution time of four applications with varying L1 cache sizes and the different schedulers. For each application, the execution time is normalized to the application's execution time with G and a 128 KB L1 cache.

Second, even with very small cache sizes, IKRA schedulers are able to provide some improvement over G—11% in SRAD v1 with just a 16 KB L1 cache. While C and R both help in these scenarios, it is F that provides the most benefit by maximizing reuse of data residing in the L1 cache.

Third, for G, in some cases, the absolute execution times *increase* in both SRAD v2 and DJ2 when the L1 cache size is increased. This is because at those sizes, the larger L1 caches result in more remote L1 hits, which take longer than L2 hits, thereby increasing the overall execution time. However, IKRA schedulers are able to effectively use the extra cache capacity, and the increased L1 cache hit rate ensures that they do not suffer from the same issue as G.

Fourth, irregular applications such as DJ2 again demonstrate the importance of work stealing. Specifically, for DJ2, regardless of the cache size, only S is able to improve upon G. The only exception is with a 32 KB L1 cache, where S is slower than G. This occurs due to graph connectivity—chunking schedules TBs with highly connected vertices (i.e., TBs with a lot of work) on the same CUs and it is not possible to steal them for load-balancing as they begin execution immediately. Since there is little to no reuse with such small cache sizes, the sub-optimal load-balance degrades performance compared to G.

Overall, IKRA schedulers are able to improve performance across a range of L1 cache sizes, even when the working set size is large compared to the size of the L1 cache.

7.2 L2-only Reuse

To separate out the benefits of exploiting inter-kernel L1 reuse, local or remote, from L2 reuse, we repeated all our experiments with GPU Coherence, the simple cache coherence protocol described in Section 2.1. This simple protocol invalidates the L1 cache at kernel boundaries and so cannot exploit inter-kernel reuse either in a local L1 or in a remote L1.[8] Figure 8 shows normalized execution time for all our regular and irregular applications with GPU Coherence.

[8]Although conceptually interesting, it is difficult to separate the impact of reuse from local vs. remote caches, since the primary mechanism of exploiting L1 reuse (seeking ownership) requires the ability to access owned data in a remote cache.

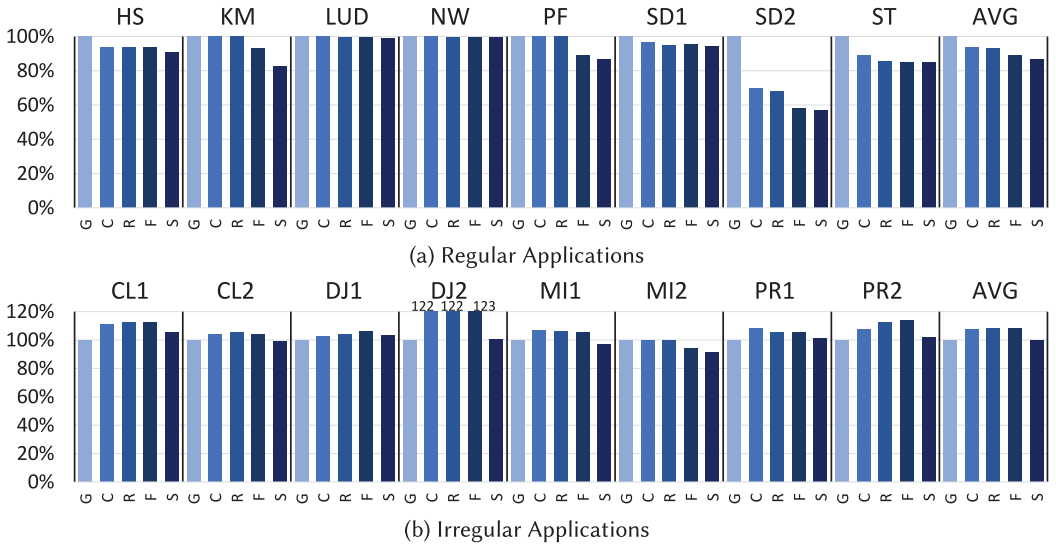

Fig. 8. Execution time for all eight regular and all four irregular applications (with two inputs each) with GPU Coherence. All bars are normalized to G.

For regular applications, IKRA schedulers are able to provide a reasonable performance improvement even with GPU Coherence; 13% less time than G on average with S. There are two sources of this performance improvement: Chunk, which provides intra-kernel reuse, and Flip, which provides reuse at the L2. Chunk provides intra-kernel reuse in a similar fashion to DeNovo. Flip with GPU Coherence, however, only increases L2 reuse and does not affect L1 reuse as DeNovo does. Consequently, the performance improvement from exploiting inter-kernel reuse alone is less with GPU Coherence than it is with DeNovo—S reduces execution time by 7% relative to C for GPU Coherence, but by 16% for DeNovo.

For irregular applications, S performs exactly the same as G on average in a system with GPU Coherence. As shown in Section 6.3, however, S consumes 10% less execution time than G when used with DeNovo. This is due to the fact that with DeNovo, IKRA schedulers are able to obtain ownership for modified variables in the L1, allowing their reuse across kernel boundaries.

Overall, we show that while our schedulers are sometimes able to improve performance by exploiting inter-kernel reuse at the L2, exploiting L1 reuse provides significant additional benefits.[9]

8 DISCUSSION

To further improve performance and energy, our IKRA scheduling algorithms can be used in conjunction with several other techniques.

- As CPUs and GPUs become more tightly integrated and kernel launch overhead becomes smaller, it is no longer necessary to only launch large GPU kernels to amortize the launch cost. In tightly integrated CPU-GPU systems, small, fine-grained kernels can be used, especially in domains such as machine learning [11]. Such kernels may have a small cache footprint, thus being more amenable to performance improvements from IKRA TB

[9]We do not perform a detailed comparison of DeNovo (Figures 5 and 6) and GPU coherence (Figure 8) in this article, as such studies have appeared in previous work [40–42]. For the applications and systems studied here, with the baseline scheduler (G), on average, DeNovo consumes 3% more cycles for regular applications and 15% fewer cycles for irregular applications compared to GPU coherence.

scheduling. In a virtuous cycle, the presence of inter-kernel data reuse support will also motivate programmers to write programs to exploit this type of reuse. Furthermore, the ability to exploit inter-kernel reuse for fine-grained kernels will motivate larger cache sizes, which are inexpensive in modern 7 nm or 10 nm technology nodes.

- IKRA TB scheduling can be combined with GPU cache bypassing techniques [9, 17, 24, 30, 31, 47, 55]. By extending the reuse criterion to inter-kernel reuse, data for which there is no reuse can be bypassed, and only reusable data can be cached. This would allow increased inter-kernel reuse, as there would be fewer evictions of reusable data.
- IKRA TB scheduling can also be used in tandem with specialized coherent structures such as stash [23]. Like a scratchpad, a stash is directly addressed and this structure may therefore be used to improve memory access efficiency in GPUs. Unlike a scratchpad, stashed data are coherent and globally addressable, so it is also possible to exploit inter-kernel reuse in the stash structure through the use of IKRA GPU scheduling.
- Although this article focuses on self-dependencies across kernel invocations, our system can be easily extended to support general affine dependencies if future applications require such support. To do so, the programmer or compiler can pass on a *dependence relation*, an affine transform describing the producer of each TB in a kernel, to the hardware. For example, $i_k \rightarrow (i + 4)_{k-1}$ represents that TB i of the current kernel depends on TB $i + 4$ of the previous kernel. A CU can then apply the transform to each TB it has just executed to obtain the new TBs it needs to execute. Using this scheme, the IKRA schedulers can exploit inter-kernel reuse for any generic affine dependency. Similarly, although we focus here on dependencies across successive kernel invocations, our system can be easily extended to exploit dependencies from a kernel k invocations ago—we simply need to track the (per-CU) history of chunk allocations and steals for the past k kernel invocations. We implemented these extensions and used them with both LUD and NW, algorithms that have *affine* non-self-dependencies in addition to self-dependencies. We found that exploiting affine non-self-dependencies in lieu of self-dependencies resulted in the same amount of reuse as just exploiting self-dependencies.
- Our schedulers can also be easily extended to support nested kernels. To do so, a chunk entry and steal queue can be kept for each level of nested parallelism (e.g., 32 in modern NVIDIA GPUs [51]). Then, whenever a child kernel is launched from a particular CU, that CU's next (lower) level of chunk entry is populated with all the thread blocks of the child kernel. Such a scheme keeps storage overhead low, does not require software changes, keeps thread block tracking simple, and still allows load balancing via work stealing (a CU can steal from any level of the remote steal queue and can optimize the stealing algorithm by targeting higher levels first).
- Our scheduling algorithms can be implemented in software, e.g., by using the framework described by Wu et al. [54], to offer extra flexibility. By using per-CU work queues for holding task information (similar to our chunk entries), the same tasks can be executed on a CU in each kernel, regardless of which TB executes them, thereby exploiting inter-kernel reuse. Reset and Flip can be implemented by simply altering the access order of the work queue, and Steal can be implemented by accessing remote work queues. A software approach, however, has two drawbacks compared to a hardware approach: (i) the work queues have to be explicitly managed by the programmer or library—our hardware approach is transparent to the programmer; and (ii) efficient atomics are required to access the work queues, but how to best implement efficient atomics in GPUs with reasonable consistency models is still a subject of research [4, 41, 42]. To the best of our knowledge, there is no prior work

on software-based inter-kernel reuse-aware thread block scheduling, and we leave a comparison of software vs. hardware implementations to future work.

- The schedulers we have proposed are representative of reasonable design points but are by no means a complete set of possible designs. For example, schedulers such as RR with reset or Steal without steal queues are also feasible. Although we would expect such schedulers to perform worse than Reset and Steal, respectively, they could present architects with more choices in the trade-off between performance and complexity. Another design dimension could be the size of each chunk in Chunk, Reset, and Flip. Instead of assigning one big chunk of TBs to each CU, several smaller chunks could be assigned on-demand. This would allow finer-grained load-balancing without work stealing in these three schedulers. However, the smaller the chunk size, the more hardware would be required to keep track of all the chunks assigned to each CU, resulting in a trade-off between performance and complexity. In practice, a one-chunk-per-CU scheme works well, as shown in Section 6.
- Finally, there is a host of other optimizations that can be used to enhance an IKRA TB scheduler, such as intra-kernel dependency tracking [49]; CTA clustering [8, 29]; co-designing a warp scheduler [26]; using dependencies to enforce synchronization [1]; performing producer-initiated prefetches to preemptively send data to a CU; augmenting profiling tools such as *nvprof* to provide scheduling metrics (e.g., reuse %) to aid the programmer in tuning their code; implementing intra-kernel locality-aware stealing algorithms for irregular applications; and adapting the large body of work on locality-aware work stealing for CPUs [3, 12, 13, 16, 37, 48, 52].

9 RELATED WORK

Related previous work on improving GPU cache utilization can be broadly divided into five groups:

Inter-kernel Reuse [51]: The work in Reference [51] has similar goals to ours, but is only applicable to nested kernels launched via frameworks such as CUDA Dynamic Parallelism (CDP), and can thus only exploit dependencies between parent and child TBs. Child TBs are scheduled on the same CU as their parent TB and are executed as soon as possible to minimize reuse distance. Furthermore, this work only rearranges *kernels*, while TBs *within* the kernels are still executed in a round-robin fashion. Rearranging TBs is more challenging due to the larger scale of the problem. Our IKRA TB schedulers are able to increase inter-kernel reuse among any pair of kernels with self-dependencies, focus on scheduling TBs rather than kernels, and perform locality-preserving work stealing. We do not perform a quantitative comparison with the work in Reference [51], since none of the applications we consider exhibit nested parallelism, and so that work is not applicable.

Intra-kernel Reuse [8, 26, 29, 49, 54]: Reference [26] uses a greedy algorithm to choose the optimal number of TBs to schedule on a CU. Additionally, it also exploits spatial locality between TBs *within* a kernel by scheduling neighboring TBs on the same CU. However, unlike our work, this technique requires changes to the warp scheduler; furthermore, it only chunks together two neighboring TBs, whereas our work chunks together significantly more while still maintaining load balance. References [8, 29] also improve intra-kernel locality via TB scheduling, but do not enable any inter-kernel reuse. Reference [54] enables software-based TB scheduling on real GPUs but neither exploits inter-kernel reuse nor considers software-based work stealing. Reference [49] provides a mechanism to use software supplied locality information in hardware, but is limited to intra-kernel locality. These techniques can be paired with ours, since we focus on inter-kernel locality while the above work improves intra-kernel locality.

Dependency-aware Scheduling [1]: This work relies upon a software-provided dependency graph to enable data-dependent parallelism and to enforce synchronization between TBs of the

same or different kernels. However, unlike our work, the dependence information is stored explicitly, occupying memory space, and is not used to exploit locality.

Kernel Scheduling [36, 51]: Reference [51] uses kernel prioritization to give TBs of particular kernels priority over TBs of other kernels. This technique aims to maximize temporal locality by scheduling dependent kernels close together in time, but does not schedule the TBs *within* these kernels, unlike our schedulers. Reference [36] uses CU partitioning to execute different kernels on different CUs. This reduces cache contention, but unlike our schedulers, it does not address TB scheduling to maximize cache reuse. Thus, our schedulers are mostly orthogonal to the above kernel scheduling techniques and can be combined with them for best performance.

Warp Scheduling [18–20, 27, 32, 38, 39, 57]: Much work optimizes how warps *within* a TB are scheduled to improve *intra*-kernel locality. In contrast, our focus is on exploiting *inter*-kernel reuse through more efficient TB schedulers. Thus, prior warp schedulers and our TB schedulers are orthogonal and combining them is an interesting direction of future work.

Cache Bypassing [9, 17, 24, 30, 31, 47, 55]: Cache bypassing has been shown to be extremely effective in GPUs. Not caching lines that will not be reused can reduce the eviction of lines that would be reused. Although bypassing can increase all types of reuse, previous work only targets intra-kernel reuse (inter-kernel reuse would require new TB schedulers).

Overall, while the techniques in this section improve cache utilization, none of them are designed to exploit inter-kernel reuse across arbitrary kernels while maintaining load-balance through locality-preserving work stealing.

10 CONCLUSION

Future GPUs need more efficient memory hierarchies to continue improving performance. We present four new GPU thread block schedulers that increase inter-kernel data reuse in the cache hierarchy. We show that for eight regular, iterative applications, static scheduling of thread blocks and the addition of simple optimizations results in average savings of 19%, 11%, and 42% in execution time, energy, and network traffic, respectively. For four irregular, iterative applications (with two inputs), we introduce locality-preserving work stealing to the thread block scheduler, to give net average savings in execution time, energy, and network traffic of 10%, 8%, and 16%, respectively. Overall, the schedulers are simple, decentralized, have extremely low overhead, do not affect the design of the rest of the GPU, and do not require software modifications.

REFERENCES

[1] Amir Ali Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi Narayan Bhuyan, and Daniel Wong. 2017. Wireframe: Supporting data-dependent parallelism through dependency graph execution in GPUs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*.

[2] N. Agarwal, T. Krishna, Li-Shiuan Peh, and N. K. Jha. 2009. GARNET: A detailed on-chip network model inside a full-system simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*.

[3] K. Agrawal, C. E. Leiserson, and J. Sukha. 2010. Executing task graphs using work-stealing. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'10)*.

[4] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. 2016. Lazy release consistency for GPUs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*.

[5] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*.

[6] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 185–195.

[7] Shuai Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*. 1–11.

[8] Li-Jhan Chen, Hsiang-Yun Cheng, Po-Han Wang, and Chia-Lin Yang. 2017. Improving GPGPU performance via cache locality aware thread block scheduling. *IEEE Comput. Archit. Lett.* PP, 99 (2017), 1–1.

[9] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive cache management for energy-efficient GPU computing. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*.

[10] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. DOI: https://doi.org/10.1145/2049662.2049663

[11] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the International Symposium on Computer Architecture (ISCA'18)*.

[12] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. 2013. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing*.

[13] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'10)*.

[14] Blake A. Hechtman, Shuai Che, Derek R. Hower, Yingying Tian, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture*. 189–200.

[15] Lee Howes and Aaftab Munshi. 2015. The OpenCL Specification, Version 2.0. *Khronos Group*. Retrieved from https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_C.pdf.

[16] Seung Jai Min, Costin Iancu, and Katherine Yelick. 2011. Hierarchical work stealing on manycore clusters. In *Proceedings of the 5th Conference on Partitioned Global Address Space Programming Models*.

[17] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2014. MRPB: Memory request prioritization for massively parallel processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'14)*.

[18] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*.

[19] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. Orchestrated scheduling and prefetching for GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*.

[20] Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*.

[21] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE Micro* 31, 5 (2011), 7–17. DOI: https://doi.org/10.1109/MM.2011.89

[22] Rakesh Komuravelli. 2014. *Exploiting Software Information for an Efficient Memory Hierarchy*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[23] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Prakalp Srivastava, Maria Kotsifakou, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: Have your scratchpad and cache it too. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*.

[24] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. 2017. Access pattern-aware cache management for improving data utilization in GPU. In *Proceedings of the International Symposium on Computer Architecture (ISCA'17)*.

[25] Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. 2016. Building heterogeneous unified virtual memories (UVMs) without the overhead. *ACM Trans. Archit. Code Optim.* 13, 1, Article 1 (March 2016), 22 pages. DOI: https://doi.org/10.1145/2889488

[26] Minseok Lee, Seokwoo Song, Joosik Moon, J. Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'14)*.

[27] Shin-Ying Lee and Carole-Jean Wu. 2014. CAWS: Criticality-aware warp scheduling for GPGPU workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*.

[28] Jingwen Leng, Tayler Hetherington, Ahmed El Tantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*.

[29] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-aware CTA clustering for modern GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.

[30] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. 2015. Adaptive and transparent cache bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'15)*.

[31] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-driven dynamic GPU cache bypassing. In *Proceedings of the International Conference on Supercomputing (ICS'15)*.

[32] Dong Li, Minsoo Rhu, Daniel R. Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S. Fussell, and Stephen W. Redder. 2015. Priority-based cache allocation in throughput processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'15)*.

[33] Sheng Li, Jung-Ho Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*.

[34] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News* 33, 4 (2005), 92–99. DOI: https://doi.org/10.1145/1105734.1105747

[35] NVIDIA. 2010. CUDA SDK 3.1. Retrieved from http://developer.nvidia.com/object/cuda_3_1_downloads.html.

[36] NVIDIA. 2017. NVIDIA Tesla V100 GPU architecture. Retrieved from https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[37] J. M. Perez, R. M. Badia, and J. Labarta. 2008. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the IEEE International Conference on Cluster Computing*.

[38] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*.

[39] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-aware warp scheduling. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*.

[40] Giordano Salvador, Wesley H. Darvin, Muhammad Huzaifa, Johnathan Alsop, Matthew D. Sinclair, and Sarita V. Adve. 2020. Specializing Coherence, Consistency, and Push/Pull for GPU Graph Analytics. Retrieved from arxiv:cs.DC/2002.10245.

[41] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU synchronization without scopes: Saying no to complex consistency models. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*.

[42] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. Chasing away RAts: Semantics and evaluation for relaxed atomics on heterogeneous systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA'17)*.

[43] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. 2013. Cache coherence for GPU architectures. In *Proceedings of the 19th International Symposium on High Performance Computer Architecture*. DOI: https://doi.org/10.1109/HPCA.2013.6522351

[44] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and WMW Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report. Department of ECE and CS, University of Illinois at Urbana-Champaign.

[45] SuiteSparse Matrix Collection. 2010. cond-mat. Retrieved from https://sparse.tamu.edu/Newman/cond-mat.

[46] SuiteSparse Matrix Collection. 2010. olesnik0. Retrieved from https://sparse.tamu.edu/GHS_indef/olesnik0.

[47] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. 2015. Adaptive GPU cache bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU'15)*.

[48] Martin Tillenius, Elisabeth Larsson, Rosa M. Badia, and Xavier Martorell. 2015. Resource-aware task scheduling. *ACM Trans. Embed. Comput. Syst.* 14, 1, Article 5 (Jan. 2015), 25 pages. DOI: https://doi.org/10.1145/2638554

[49] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B. Gibbons, and Onur Mutlu. 2018. The locality descriptor: A holistic cross-layer abstraction to express data locality in GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA'18)*.

[50] Virtutech. 2006. Simics full system simulator. Retrieved from http://www.simics.net.

[51] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2016. LaPerm: Locality aware scheduler for dynamic parallelism on GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA'16)*.

[52] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu. 2014. Optimizing load balancing and data-locality with data-aware scheduling. In *Proceedings of the IEEE International Conference on Big Data (Big Data'14)*. 119–128.
[53] WikiChip. 2019. Exynos 9820 - Samsung. Retrieved from https://en.wikichip.org/wiki/samsung/exynos/9820.
[54] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the International Conference on Supercomputing (ICS'15)*.
[55] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *Proceedings of the 21st IEEE Symposium on High Performance Computer Architecture (HPCA'15)*.
[56] Amir Yazdanbakhsh, Gennady Pekhimenko, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C. Mowry. 2019. *Towards Breaking the Memory Bandwidth Wall Using Approximate Value Prediction*. Springer International Publishing, 417–441.
[57] Z. Zheng, Z. Wang, and M. Lipasti. 2014. Adaptive cache and concurrency allocation on GPGPUs. *Comput. Archit. Lett.* PP, 99 (2014), 1–1. DOI: https://doi.org/10.1109/LCA.2014.2359882

Received June 2019; revised June 2020; accepted June 2020