PAVER: Locality Graph-based Thread Block Scheduling for GPUs

DEVASHREE TRIPATHY, University of California, Riverside, USA
AMIRALI ABDOLRASHIDI, University of California, Riverside, USA
LAXMI NARAYAN BHUYAN, University of California, Riverside, USA
LIANG ZHOU, University of California, Riverside, USA
DANIEL WONG, University of California, Riverside, USA

The massive parallelism present in GPUs comes at the cost of reduced L1 and L2 cache sizes per thread, leading to serious cache contention problems such as thrashing. Hence, the data access locality of an application should be considered during thread scheduling to improve execution time and energy consumption. Recent works have tried to use the locality behavior of regular and structured applications in thread scheduling, but the difficult case of irregular and unstructured parallel applications remains to be explored.

We present PAVER, a priority-aware vertex scheduler, which takes a graph-theoretic approach towards thread scheduling. We analyze the cache locality behavior among thread blocks (TBs) through a just-in-time (JIT) compilation, and represent the problem using a graph representing the TBs and the locality among them. This graph is then partitioned to TB groups that display maximum data sharing, which are then assigned to the same SM by the locality-aware TB scheduler. Through exhaustive simulation in Fermi, Pascal and Volta architectures using a number of scheduling techniques, we show that PAVER reduces L2 accesses by 43.3%, 48.5%, 40.21% and increases the average performance benefit by 29%, 49.1%, 41.2% for the benchmarks with high inter-TB locality.

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**.

Additional Key Words and Phrases: GPGPU, Thread Block, Dependency Graph, locality

1 INTRODUCTION

The massive parallelism provided by general-purpose GPUs (GPGPUs) is in demand from many areas of industry and research. Possessing numerous compute cores in their streaming multiprocessors (SMs) and enormous memory bandwidths as high as 1555 GB/s [41], GPGPUs have become the de-facto accelerator of choice in many scientific domains. To support the complex memory access patterns of applications, GPGPUs have a multi-level memory hierarchy consisting of an L1 data cache private to each SM, a banked shared L2 cache connected through an interconnection network across all SMs and high-bandwidth banked DRAM.

With the amount of parallelism GPUs can provide, memory traffic becomes a major bottleneck for present-day GPUs, mostly due to the small amount of private cache that can be allocated for each thread, and the constant demand of data from the GPU's many computation cores. With the ever-increasing data size of GPU applications, and each thread having to process more data, simply increasing the cache sizes is not a viable option, since the additional area will incur extra cost and overhead. This means that smaller L1 and L2 caches are much more likely to suffer from cache thrashing, i.e. eviction of cache lines which could have been used by other functional units. Cache thrashing can lead to more cache operations, which means more energy consumption, and tremendous under-utilization of the other GPU resources, resulting in under-performance

Authors' addresses: Devashree Tripathy, University of California, Riverside, 900 University Ave, Riverside, CA, 92521, USA, devashree.tripathy@email.ucr.edu; AmirAli Abdolrashidi, University of California, Riverside, 900 University Ave, Riverside, CA, 92521, USA, amirali.abdolrashidi@email.ucr.edu; Laxmi Narayan Bhuyan, University of California, Riverside, 900 University Ave, Riverside, CA, 92521, USA, bhuyan@cs.ucr.edu; Liang Zhou, University of California, Riverside, 900 University Ave, Riverside, CA, 92521, USA, lzhou008@ucr.edu; Daniel Wong, University of California, Riverside, 900 University Ave, Riverside, CA, 92521, USA, danwong@ucr.edu.

[11, 23, 56, 62, 63]. To minimize this, all threads need to utilize the shared cache memory spaces with each other as efficiently as possible.

The most common relationships between the threads sharing the cache are read-after-write (RAW) and read-after-read (RAR) cases. RAW constitutes of data dependency among tasks, e.g. thread blocks (TBs), also known as *structured parallelism*. In this case, a certain execution order among the tasks will be formed and the execution time will be bound to a *critical path*. Exploiting this order can improve the performance substantially. There have been numerous works on structured parallelism in CPUs [3, 12, 14, 51], and more recently in the realm of GPUs [2, 6, 16, 60]. In particular, they try to exploit the data locality between parent and child TBs.

RAR, or local data sharing, on the other hand, happens in *unstructured parallelism*, in which the tasks are similar and independent, and thus free to be executed in any order. Its impact is more prominent when there is no write-allocate policy in place. As a result, the program's outcome would be correct regardless of task ordering. However, processes can still subtly affect each other in terms of shared resources and, subsequently, the performance, in contrast to the more explicit sharing in structured parallelism. To make better use of the cache data, the data locality of the parallel-run tasks must be observed and considered with respect to a given cache architecture. Research works have addressed this issue in the multi-core CPU area [20, 61], and Wang et al [60] improved cache performance with respect to data-reuse involving parent-child thread blocks in GPUs. To the best of our knowledge, however, ***PAVER*** (**P**riority-**A**ware **V**ertex schedul**ER**) is the first work on exploiting data locality in unstructured parallelism in GPUs.

At the TB level, there have been attempts to take advantage of locality based on a specific data access pattern [22]. In [29], Lee et al. propose Block CTA Scheduling (BCS) which naively assigns two consecutive TBs to the same SM. However, their approach works only for row-major applications, i.e. applications optimized to run with row-major data structures, such as matrix multiplication, n-body, and hotspot. Since the grid structure of the tasks is application-specific, PAVER addresses this problem with a generic graph-based approach to improve performance and memory efficiency. We do so by creating a graph of TBs using their data sharing statistics, where the vertices represent the TBs and weighted edges between TBs denote the number of shared data locations.

Designing a graph structure requires us to know the access footprint per TB in the application a priori and then deciding which TBs to group in the same SM to maximize the cache utilization. Hence, it is necessary to analyze the cache access characteristics before the execution. There has been substantial work on CPU data access profiling using compilers in the past [4, 20, 58, 64, 65]. Such profiling work has also been done for GPUs [13, 53]. Research has started recently to speed up the GPU profiling through sampling and other techniques [20]. However, profiling requires the user to run the code in order to detect opportunities to improve the performance [17]. Compiler-assisted methods can extract information directly from the code itself e.g. static compiler analysis, which can be used to optimize cache bypassing, warp scheduling and thread throttling [23, 24, 32, 47]. In PAVER, we propose a just-in-time compilation approach to gather the data sharing statistics among the thread blocks which constitute the same kernel and are able to use the same allocated memory. The JIT analysis is accomplished after compilation and before the kernel launch [40].

We partition the graph in order to assign TB groups with the most locality to SMs. We explore various graph partitioning policies, such as k-way and recursive bi-partitioning algorithms based on METIS graph partitioning software [21], and Prim's maximum spanning tree (MST)-based algorithm [46]. The partitioning is stopped when the maximum number of TBs is allocated to an SM, as determined by the application and resources. To improve load balancing between SMs, we also incorporate a task-stealing process to move a TB from one SM queue to another when all the TBs in the latter finish early. This ensures a decent load balance in the final phase of the execution.

In short, PAVER presents a novel graph-theoretic approach for TB scheduling on GPUs based on the cache sharing behavior between the TBs. Our partitioning algorithm ensures maximum cache sharing within an SM for L1 locality, high L2 locality among the SMs, and also ensures load balancing between the SMs. In a SM, maximum allowed concurrent TBs are sent for execution in the form of bunches of 32 threads each, called a warp, to a warp scheduler. This scheduler checks for ready warps within the pool of available warps and once a TB has finished execution, another TB assigned to that SM starts executing. There are several warp scheduling policies with the baseline policy as Greedy Then Oldest (GTO) in which a single warp is prioritized for execution until it hits a long latency operation after which it is replaced by the warp which was assigned to the core for the longest period of time.

The following outlines the organization of this paper:

- In Section 2, we explore the baseline architecture and show how locality awareness can benefit the application performance.
- In Section 3, we generate the *TB locality graphs* of various benchmarks through analyzing memory access behaviour of individual thread blocks. We also present details of the JIT compiler analysis, adopted in this paper.
- In Section 4 we design TB scheduling policies, starting from MST, and improve it to k-way partitioning and recursive graph bi-partitioning. We compare these partitioning techniques to understand its effect on L1 and L2 locality.
- In Section 5 we explain the PAVER runtime and TB scheduling and how task stealing can help with an application's load balancing. We further discuss the architectural support to store the TB-groups produced by various TB-partitioning strategies and using them to guide TB scheduling.
- In Section 6, we evaluate PAVER and show that our proposed scheduler can achieve average speedup of 29%, 49.1%, 41.2% compared to the baseline scheduler LRR in Fermi, Pascal and Volta architectures. Benchmarks selected from widely used benchmark suites Parboil [54], Rodinia [7], Polybench [45], ISPASS [1], and CUDA SDK [36] were analyzed and classified into high, low and no inter-TB locality categories.

2 BACKGROUND AND MOTIVATION

TB scheduling is managed by the GigaThread engine in Nvidia GPUs [30]. Despite efforts to approximate the behavior of the TB scheduler [34], there are few official details on it. The baseline TB scheduler is assumed to be using a loose round-robin (LRR) policy, as empirically observed in prior work [31]. A round-robin policy implicitly takes advantage of locality among consecutive TBs that can simultaneously access the L2 cache. To capture both L1 and L2 locality, some prior works assign TBs at the granularity of a group of consecutive TBs (typically groups of two) to an SM [29, 55]. These prior techniques specifically exploit the two-dimensional data grid with locality typically occurring between TBs in the same row. Such a policy is bound to fare poorly if there is large column-wise communication, or even worse, if the communication between TBs is arbitrary.

As a novelty, we attempt to establish an order in thread block execution by using a graph-based approach to maximize data locality. This method is completely generic and is able to extract locality patterns through graph-theoretic approaches regardless of the application's data layout or algorithmic behavior. To this end, we determine the locality among all the thread blocks per kernel before the kernel launch, and create a weighted graph pertaining to that kernel (for multi-kernel applications, a graph is generated for each kernel). The vertices are annotated with thread block IDs and the edge weights represent the number of shared data references between two TBs. The higher the weight of an edge between two nodes, the higher is the locality between the two TBs.

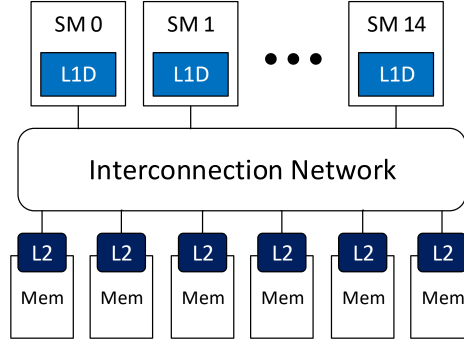

Fig. 1. GPU memory hierarchy

We will later explore various locality graph partitioning techniques in order to assign TBs to SMs to maximize cache reuse.

2.1 Baseline GPGPU architecture

In this work, we use Nvidia GTX480 (Fermi [37]), Nvidia TITANX (Pascal [38]) and Nvidia TITANV (Volta [39]) architectures for evaluation purposes. Our technique exploits application-level characteristics and can be generally applicable to all architectures. Table 1 describes the specifications of GTX480, TITANX and TITANV, and Figure 1 depicts the memory hierarchy layout for a generic GPU. As it can be seen, every streaming multiprocessor (SM) has an L1 cache, and there is an L2 cache per memory channel that is shared by all the SMs. Whenever a load operation attempts to access a data which is already cached, it is considered a *cache hit*. Otherwise, it results in a *cache miss*, and the cache will request the data from the lower memory in the hierarchy. A miss can occur due to the data simply never being present (cold miss), or due to the data block being evicted from cache because of cache size limit (capacity miss), or set conflicts (conflict miss), or becoming stale after another 'sibling' cache has modified it. In Fermi, however, since there is a write-evict policy [42], we do not experience cache invalidations due to coherence protocols.

Table 1. Fermi, Pascal, Volta GPU specifications (for evaluation)

Architecture	Fermi	Pascal	Volta
# of SMs	15	28	80
Max # of TBs/Warps/Threads per SM	8/48/1536	32/32/2048	32/64/2048
L1/Shared Mem. Cache Size per SM	16/48 KB	48/96 KB	32/96 KB
Total L2 Cache Size	768 KB	3MB	4.5MB
Core Frequency	700 MHz	1 GHz	1.2 GHz

All load/store units (LDST) in the SMs have a memory coalescing unit to reduce unnecessary cache accesses. To further accommodate memory coalescing and avoid extra memory traffic, an SM's L1 data cache has a miss status holding register (MSHR) table, which sends the request to the L2 cache and holds the pending data block request while it is being loaded. If another thread tries to access the same block again, it is called a *hit reserve* or MSHR hit. In this case, the operation will wait for the request in the MSHR to be finished. There can be MSHR hits in a cache only if there is data sharing across warps. If the MSHR is full, any more requests will be a miss and named a *reservation fail* [5].

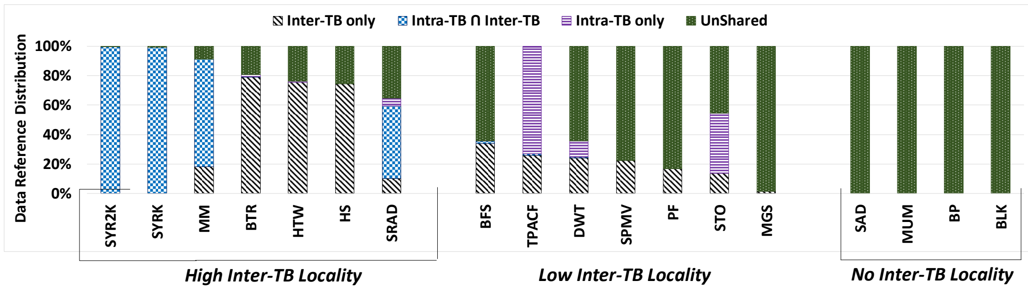

Fig. 2. Data reference sharing distribution

2.2 L1/L2 hit and miss distribution

There are two main types of locality: temporal and spatial. Temporal locality refers to the same data in the cache used at different times, while spatial locality involves different units accessing different parts of the same data block, e.g. two TBs accessing two adjacent elements in the same cache line. We define a term called *"block locality"* to capture the usage frequency of any data element inside a data block either due to temporal or spatial locality during the execution. If the GPU knows which data block is needed by which specific TB, it can schedule the appropriate TB before the data block is evicted. GPU execution can benefit from a planned scheduling, which maximizes data locality in all caches. If TBs that use distant parts of the memory also share the same SM, it will lead to unnecessary L1 evictions and thrashing, hurting the performance. Similarly, if TBs executing on different SMs lack block locality, they will suffer from L2 eviction.

We measured the L1 data cache and L2 cache hit and miss distribution for different benchmarks and observed that on an average, 33.6% of all misses in L1 are conflict or capacity misses. More conflict or capacity misses lead to more frequent L2 cache accesses, which may in turn result in L2 evictions. Also, it was noted 9% conflict + capacity misses in L2 on average. Our method will reduce the average cache miss rate due to both conflict and capacity misses.

Data Reference Distribution: Figure 2 shows the data reference distribution for different benchmarks. The y-axis shows the normalized total number of global read accesses in an application as a percentage. We categorize data accesses into the following type of references:

- Unshared - single warp in a TB accesses a data ;
- Intra-TB - multiple warps within a TB access same data ;
- Inter-TB - multiple TBs access same data ; and
- Intra-TB ∩ Inter-TB - data block is accessed by multiple warps within a TB, and across multiple TBs.

As it can be seen, the benchmarks SYRK, SYR2K, MM, BTR, HTW, HS, SRAD have heavy inter-TB data sharing; BFS, TPACF, DWT, SPMV, PF, STO and MGS have low inter-TB sharing; SAD, MUM, BP and BLK have no inter-TB data sharing.

Therefore, if TBs sharing the data are assigned to the same SM, we can increase the L1 hits, thereby reducing the number of L2 accesses and lowering L2 conflict misses, and improving IPC. On an average, 27% of data sharing between TBs is observed. Hence this paper targets exploiting inter-TB and intra-TB ∩ inter-TB references by cleverly scheduling the TBs instead of the naive LRR TB scheduler.

3 GENERATING LOCALITY GRAPHS

3.1 PAVER overview

Figure 3 overviews the **PAVER** framework, a **P**riority-**A**ware **V**ertex schedul**ER**. It is a locality-aware thread block scheduling (TB) framework, which is guided by locality graph analysis. The load address ranges for a TB are extracted from the PTX code. The locality graph is constructed from the extracted locality information, where the vertices of the graph represent the TB ID and the edge weight represents the number of common shared data-references accessed by those TBs. The TB graph capturing the inter-TB locality is partitioned such that each partition contains TBs having maximum locality among them (explained in Section 4). After this stage, we enforce the execution of a TB partition in an SM through hardware support. A locality-aware TB scheduler assigns the TBs to the SMs in the order specified by the partitioning algorithm (explained in Section 5).

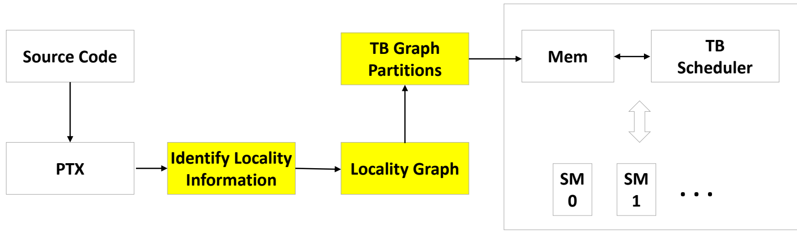

Fig. 3. PAVER Overview: Paver generates the locality graphs by identifying the locality information in the PTX code at JIT. The TB-graph partitions are then fed to TB scheduler at run-time to leverage the inter-TB data locality.

3.2 Locality Graph:

The data reference sharing information is used to generate a locality graph in the form of an adjacency matrix. In this matrix, every element in location (i, j) specifies the number of data references shared between TB_i and TB_j. We represent the graph by an adjacency matrix is symmetrical around the diagonal (undirected graph). The examples of an adjacency matrix and the corresponding locality graph are shown in Figure 4. The nodes are numbered as per the TB ID, and node weights represent the number of instructions executed by the node. In our case, the weights of all nodes are the same, because all the TBs execute the same number of static instructions in an SIMD manner.

Figure 5 visualizes the adjacency matrix representing the locality graph for different applications. Both X and Y dimensions represent the TB numbers. The sharing (edge) between them is represented by a point in the figure, where the color density of the point represents weight on the particular edge. It may be observed that maximum sharing between two TBs may occur when they are adjacent (MGS, STO, PF, HTW) or when they are far away (BFS, BTR, SPMV). Though the prior work in [29] assumes that consecutive TB have max locality, this is not necessarily true. Sometimes there is sharing among adjacent TBs in the same row and same column of the 2D Grid (SYRK, SYR2K, MM), same row and first column of 2D Grid (HS, DWT, SRAD), same row of 1D Grid (MGS, STO, PF, HTW) or arbitrary (BFS, BTR, SPMV). A generic graph representation accounts for all these cases.

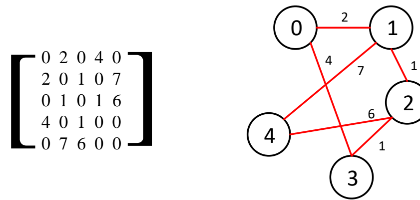

Fig. 4. An example of an adjacency matrix and the corresponding locality graph

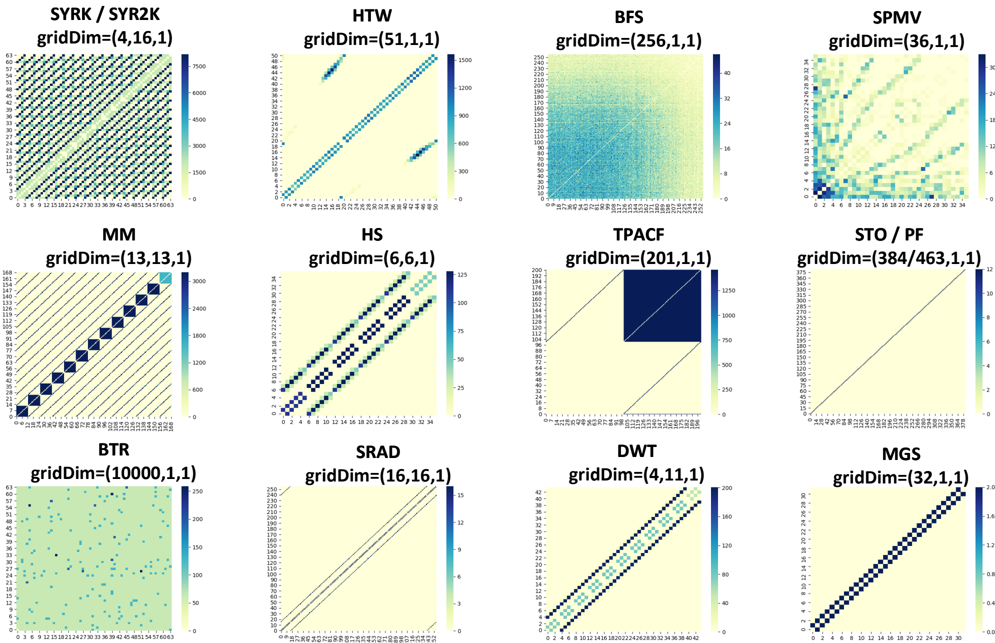

Fig. 5. Adjacency matrix of Various Applications.

3.3 Identifying Locality Information

In order to run CUDA applications, the CUDA code must first be parsed into the PTX intermediate representation (IR) during compilation. In this stage, the code is converted into a quasi-Assembly structure, with instructions using input/output registers, indirect addressing, etc. However, after conversion to PTX, the code is still not ready for execution on real hardware, as GPUs require a code format specific to their architecture, i.e. the SASS representation. The conversion from PTX to SASS is performed via just-in-time (JIT) compilation at the time of application load [40]. This is when some of the remaining unknown parameters are resolved that might be dependent on user input during the kernel launch in order to specify the kernel's characteristics, such as input/output arguments and pointers, grid and block sizes, etc. When the kernel is converted to SASS format with all the necessary parameters, only then can it run on the target GPU. In our work, we aim to perform analysis on the PTX code before the kernel launch in order to extract locality information from the kernel PTX.

Profiling has been extensively used in CPUs [4, 58, 65]. In a recent paper for GPUs [13], Ocelot [10] was used to instrument the PTX code. The instrumentation involves inserting a device function call to gather the memory trace of the entire program in order to detect the uncoalesced accesses in the code. Alternatively, Shen et al. [53] use an instrumentation engine, built on top of LLVM [28] to place bits of code on both the host and device sides to track statistics such as memory reuse distance. The code will then be translated into PTX. However, profiling requires the user

to run the application once to extract all the relevant information, which is not desirable if the data size is large. To minimize the user burden, preprocessing the code before execution is needed. In PAVER, we propose a JIT-based static analysis to extract TB locality information from each kernel. In a CUDA application, inputs and outputs are given to the CUDA kernel call as base pointers initialized through `cudaMalloc()`. The kernel would then read the data from the input data structures and write the results in the outputs. In order to generate a locality graph for our application, the accessed read addresses will need to be extracted in order to determine address in the global memory, accessed by a TB.

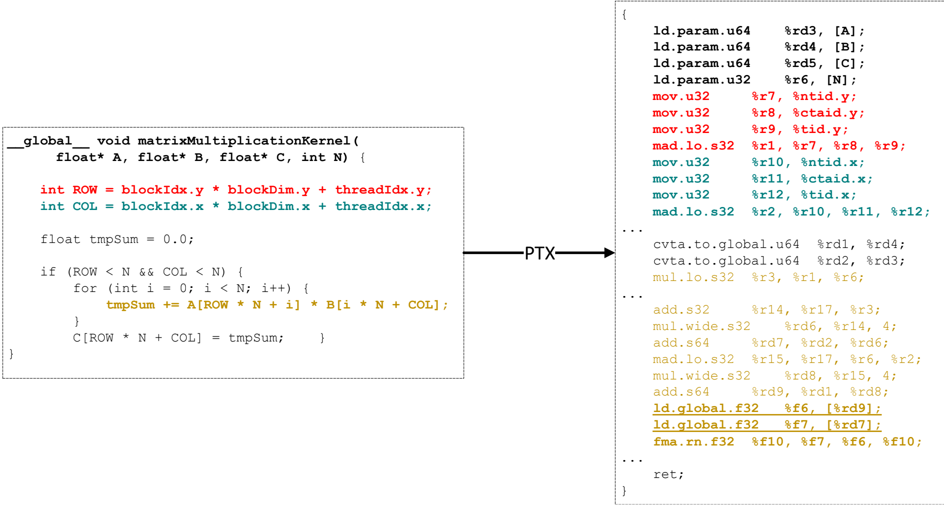

Fig. 6. Matrix multiplication source code (left) and some of its corresponding PTX representation (right)

Figure 6 shows an example of the matrix multiplication code in CUDA converted into PTX. Some of the corresponding codes on both sides have been highlighted with matching colors. It is in our interest to extract the information from the global memory read instructions (e.g. `ld.global`, underlined) for each TB in order to determine the locality among all TBs in the kernel. In order to extract memory access information, a JIT analysis is necessary, since the arguments in a kernel call may not be known until the kernel is finally being called. In addition, some of the kernel parameters, such as kernel grid size, block size and input data size, may also be unknown before the call, e.g. if they are dependent on user input. However, once the kernel is called, all the information stated above will be available. Therefore, each thread's unique parameters, such as thread ID and block ID, would also be known at that time, and would remain the same throughout the kernel's execution, which simplifies our analysis. Once the analysis is complete for the called kernel, we have each TB's memory access locations, which can then be used to construct a locality graph. The exact values of the matrix size N and the base addresses of input/output matrices A, B, C are known after *malloc* in GPU. When the grid size and TB size become available, so do the ranges of the existing thread-specific values, namely *threadIdx* and *blockIdx*. Therefore, *ROW* and *COL* (colored red and teal) expressed in terms of these two values can readily be determined for each thread at JIT compilation. Also, we can locate the global memory access instructions and identify which elements of the arrays are accessed by each TB, thereby, determining the value range of the memory accesses per TB. Here, for example, with *ROW* and N being known and i iterating from 0 to N, we know all the possible values of index $ROW * N + i$, and thus all elements of

Table 2. Data reference sharing across TBs for various applications from [1, 7, 36, 54].

Benchmark	Description	*SpScore*	Total TB	Total data references	Shared TB	Degree of sharing
HS	Hotspot	0.995823097	1849	524288	1849	0.009194
PF	Pathfinder	0.995689675	463	2100000	463	0.036717
STO	StoreGPU	0.99480523	384	49164	384	0.044271
BFS	Breadth first search	0.006408691	3907	7811036	3907	0.066406
SRAD	(Speckle Reducing Anisotropic Diffusion	0.977020264	16384	4196352	16384	0.066406
TPACF	Two Point Angular Correlation Function	0.745055815	201	148388	201	0.084577
MM	Matrix-multiply	0.857988166	169	1024	169	0.100592
SYR2K	Symmetric rank-2k	0.4140625	256	196608	256	0.265625
SYRK	Symmetric rank-k	0.4140625	256	131072	256	0.265625
HTW	Heart Wall	0.923875433	51	78135	51	0.333333
MGS	Merge Sort	0.939453125	32768	8454144	32768	0.375
BTR	B+ Tree	0.000167	10000	674287	10000	0.0017
DWT	Discrete wavelet	0.862603306	4096	65536	4096	0.386364
SPMV	Sparse-Matrix Dense-Vector Multiplication	0.430555556	765	6981392	765	0.472222
MUM	MummerGPU	1	196	1055946	0	-
SAD	Sum of Absolute Differences	1	1584	25344	0	-
BLK	Black scholes	1	480	6000000	0	-
BP	Back-propagation	1	4096	1114112	0	-

A being read in the kernel by a particular thread. Similarly, all the elements of matrix B and the thread blocks accessing them can be known. Any $TB_k \in \{0, 1, ..., gridDim.x * gridDim.y * gridDim.z\} (= U_{TB})$ accesses a set of elements in matrices A and B. For A and B, the access sets for TB_k will be $A_k = \{i + N(\lfloor \frac{TB_k}{gridDim.x} \rfloor blockDim.y + j) \mid i \in [0, N), j \in [0, blockDim.y)\}$ and $B_k = \{(i.N(TB_k \% gridDim.x) blockDim.x + j) \mid i \in [0, N), j \in [0, blockDim.x)\}$. Using this, we can obtain the common set of elements in matrices accessed by every TB_i and TB_j, denoted by sets (A_i, B_i) and (A_j, B_j). Therefore, the number of common data elements accessed by TB_i and TB_j in the locality graph will be:

$$L(TB_i, TB_j) = |A_i \cap A_j| + |B_i \cap B_j|$$

Table 2 showcases the characteristics of the benchmarks used in this work. Note that *Sparsity score* is expressed as: $SpScore = 1.0 - \frac{non\text{-}zero\ elements\ in\ matrix}{total\ elements\ in\ matrix}$. *Shared TB* refers to the number of TBs, which share at least one data reference.

The reported statistics are averaged over all the kernels for an application. It may be noticed that the *degree of sharing* (ratio of shared blocks to shared TBs) varies widely depending on the application. The applications with large sharing are likely to benefit from proper TB scheduling.

In PAVER, our focus is *static memory analysis* at JIT, or analysis of memory locations available before the execution of the kernel, such as device variable addresses, immediate values, and kernel parameters. As an example, if `A` is an input kernel argument, it counts as a static memory location if we use an index that is available at JIT, such as `A[0]`, `A[i]` (where `i` is a loop parameter), `A[tid]`, etc. At this time, we do not analyze *non-static memory locations*, which can only be known during run-time, e.g. pointer chasing. An example of a non-static memory location is `A[A[0]]`, since the value stored in `A[0]` cannot be known except at run-time, which is outside the scope of this work.
Overhead: The JIT analysis is done before the kernel-launch and hence does not affect the kernel-execution time, but it increases the *host side time*. The functions like initialization, memory allocation (malloc), cudaMalloc, cudaMemcpy, defining the gridDim and blockDim contribute

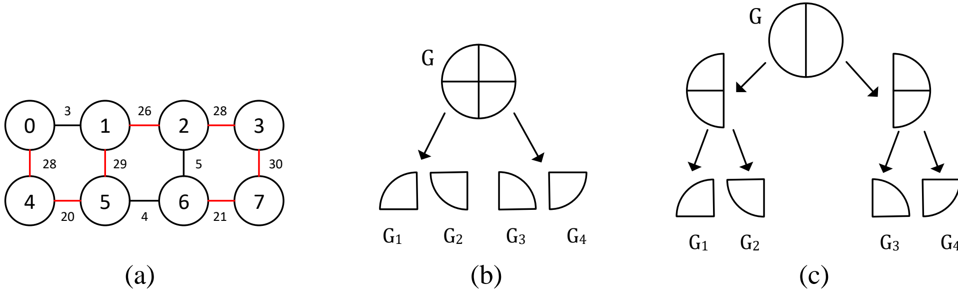

Fig. 7. Overview of TB grouping and ordering approaches. (a) maximum spanning tree. (b) k-way partitioning. (c) recursive bipartitioning.

towards the host side time. Usually, the kernel load is executed in CPU before the GPU execution starts and is negligible. Since the JIT analysis is done right after the grid and block dimensions are defined, the JIT overhead is calculated as : $Overhead = \frac{JIT\ time}{host\ side\ time\ in\ baseline+JIT\ time} * 100$. The JIT overhead for some applications with high inter-TB locality SYRK, SYR2K and MM are 0.26%, 0.2% and 0.01%, respectively. Note that device to host cudaMemcpy has been excluded from the overhead calculation as it does not delay the kernel load time. The overhead will reduce further if the device to host cudaMemcpy time is included.

4 PAVER THREAD BLOCK (TB) SCHEDULING

In this section, we explore three different graph-based locality analysis techniques to partition and group thread blocks to guide TB scheduling, namely; a naive approach using maximum spanning tree (MST); a k-way partition-based method to improve the hit rate of L1 caches; and a recursive bipartitioning-based approach to account for both L1 and L2 cache performances. Figure 7 shows an overview of each grouping strategy. TB grouping and ordering is done at the just-in-time (JIT) compilation and then the TB partitions are passed onto the GPU's global memory to guide TB scheduling.

4.1 Maximum Spanning Tree-based TB Scheduler (MST-TS)

In our first approach, we map this problem into a variant of the traveling salesman problem (TSP), which aims to traverse all the vertices (TBs) in the graph with minimum (or maximum) traveling cost. In our case, we can leverage the TSP problem to capture all the significant cases of data sharing in terms of maximum edge weights in the graph G.

The heuristic we use to solve TSP is the maximum spanning tree (MST). The MST could be constructed using either Prim's [46] or Kruskal's algorithm [26]. In this work, we use Prim's MST.

Once the MST is constructed, we have a path connecting all the TBs. An example of an MST solution (red lines) is displayed in Figure 7 (a), where each node represents a TB. We then partition consecutive TBs in the MST into N groups of size x, where x is equal to the number of TBs that can run concurrently in the SM and N is the number of SMs in the GPU architecture. N is limited by the hardware resources (registers, shared memory, number of threads, maximum number of TBs in an SM, etc.). After kernel launch, the first TB-group of size x is assigned to a SM, thereafter, the subsequent TB groups assigned to the SM have one TB each. For example, if our N is 2 and x is 2, then we have 6 groupings of (0,4), (1,5), (2), (3), (7) and (6). This assignment aims at achieving high L1 locality and load-balancing across the SMs.

This approach captures more inter-TB locality than BCS and LRR. BCS groups two consecutive TBs into a pair and assigns them to the same SM. This approach will not work for the applications having column wise locality in a 2D grid or arbitrary locality pattern. Additionally, the TB pair assignment is delayed till the TB contexts for the pair is available, leading to resource starvation in the SM. MST-TS overcomes the SM-under utilization issue observed in BCS by assigning a TB as to the SM as soon as the context becomes available and using a graph-based representation which accounts for all types of locality patterns.

However, it only captures one-dimensional sharing without considering data sharing between more than one TB. Thus, we need a more generalized approach.

4.2 *k*-way Partition-based TB Scheduler (Kway-TS)

Although the MST ordering could enhance the data locality within TB groups, it considers a TB that has the highest data sharing on the path and ignores other connections to a TB that may also share data. Given a graph $G\ (V, E)$, E' is the set of edges belonging to the MST. In the MST ordering, we only order the TBs based on the data sharing on E'. A partition based on the complete edge set E should produce better inter-TB locality-aware groupings for L1 cache locality inside the SM.

We present k-way partitioning, in which k is the total number of partitions equal to the number of SMs. We evenly partition the entire set of TBs to all the k SMs considering data sharing. For example in Figure 7 (b), we partition the graph G into 4 groups to be assigned to 4 SMs. Graph-partitioning tools, such as METIS [21] or Chaco [15] can be used, to partition the graph in a load-balanced manner while maximizing the sum of edge weights within partitions. In this work, we utilize METIS for graph partitioning. The main advantage of this method over MST-ordering is that the graph is partitioned in a way that the groups have the highest connectivity, i.e. TBs have the highest locality within a group. Therefore, it leads to a much higher L1 locality. Since all the TBs in a partition cannot execute concurrently due to resource limitation in an SM, *we re-order the TBs in each partition using Prim's MST such that the subset of TBs in each partition executing concurrently has maximum locality with each other.*

One disadvantage of k-way partitioning is that L1 data locality is maximized within each partition. However, each partition is executed in an SM over the entire kernel runtime, with multiple partitions without locality running simultaneously on different SMs. The locality between the SMs may be lost leading to L2 thrashing. This type of scheduling prioritizes L1 locality over L2 locality.

4.3 Recursive Bi-partition-based TB Scheduler (RB-TS)

The problem mentioned before with k-way partitioning inspires us to design the recursive bi-partitioning scheduling for TBs to guarantee maximum data locality across L1 and L2 and load balance between TB groups. As shown in Figure 7 (c), we recursively partition the graph G into two parts until the partition size is less than the maximum allowed TBs in each SM (which we denote as x from Section 4.1). This essentially creates a binary tree where the leaf nodes (G_1, G_2, G_3 and G_4) are prioritized from left to right. This preserves L1 locality by TBs grouped within the same leaf node, and preserves L2 data locality by concurrently scheduling adjacent groups on different SMs that share the same parent.

The pseudocode for our recursive bipartitioning algorithm is shown in Algorithm 1. Q is the queue to store the TB groups, which need further partitioning. L stores all the final leaf TB groups in the partition tree. At every iteration, we pop out one sub-graph G_i from the queue and partition it into two different TB groups. If G_i is smaller than the TB capacity of an SM, we get one final TB group. Otherwise, the algorithm pushes it to the back of the queue Q and waits for further bipartitioning. Our algorithm uses METIS to achieve recursive bipartitioning with load balancing.

Algorithm 1: Recursive bipartitioning

```
1  Let Q be the queue of TB groups
2  Let L be the list of TB groups for SM scheduling
3  Q.push(G_0)
4  while Q not empty do
5      G_i =Q.front()
6      Q.pop()
7      (G_i^0, G_i^1)=METIS.partition(G_i, n = 2)
8      if G_i^0.size() < maxTB then
9          move G_i^0 to list L
10     else
11         Q.push(G_i^0)
12     end
13     if G_i^1.size() < maxTB then
14         move G_i^1 to list L
15     else
16         Q.push(G_i^1)
17     end
18 end
```

5 PAVER RUNTIME

In this section, we will discuss the generalized PAVER Runtime, which schedules thread blocks (TBs) based on graph-based TB grouping strategies. Once the TB groups have been created and re-ordered at JIT compilation, the TB scheduler uses them at runtime to schedule TBs among SMs. For PAVER TB Scheduling policies, a global queue (located in global memory) is used to store the pointers to the TB groups. We assume that the maximum number of concurrent TBs executing on an SM is x, which is dependent on the kernel resource requirement like registers, shared memory, local memory and number of threads in an TB.

5.1 Hardware Implementation

The architectural modification needed for supporting PAVER is shown in Figure 8. Once the kernel is loaded, the TB groups and ordering within each group are stored in the global memory as an array of arrays. The *global queue* stores the array of partitions. Each entry in the array corresponds to a partition and points to the head pointer of an array that stores all the TB groups of that partition in order. The number of TB groups and size of each TB group differs by the partitioning technique used. Each SM is associated with two registers (*next, tail*) which point to the TB group's *head* (initially) and *tail* assigned to that SM, respectively. Another 2-byte register in the SM stores the *next TB ID* i.e. TB group[*next*]. Once the current TB from the TB group is issued to the SM and starts executing, the *next* register value is updated to point to next TB in the TB group and the next TB register is loaded with the new value. This next TB register guides the thread block scheduler.
Storage Overhead: The 3 extra registers per SM to store *next, tail and nextTB* incur an area overhead of 64-bit * 2 (for next and tail) + 16-bit (for nextTB) i.e. 18 Bytes. So, total storage overhead for Fermi (15 SMs), Pascal (28 SMs) and Volta (80 SMs) are 270 Bytes, 504 Bytes, and 1440 Bytes respectively. This overhead is negligible compared with the area of other on-chip storage structures in Fermi/Pascal/Volta GPU such as L1 cache (240KB/720KB/480KB), shared memory (1920KB/3840KB/3840KB) and Register File (1920KB/14MB/40MB).

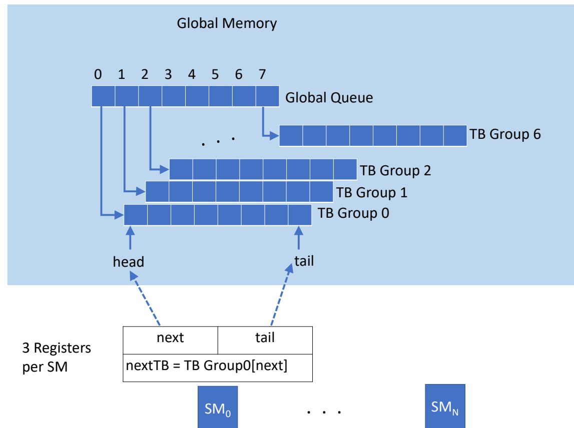

Fig. 8. Storing TB Groups in Global Memory: Once the TB-Groups are generated by different graph-partitioning strategies (MST, Kway, Recursive Bipartitioning), they are stored in the global memory. A global queue (located in global memory) is used to store the pointers to these TB groups. Each SM is associated with two registers (*next, tail*) which point to the TB group's *head* (initially) and *tail* assigned to that SM, respectively. Once the current TB from the TB group is issued to the SM and starts executing, the *next* register value is updated to point to next TB in the TB group and the next TB register is loaded with the new value i.e. TB group[*next*]. This next TB register guides the thread block scheduler.

Timing Overhead: Any timing overhead would occur when the nextTB is yet to be loaded from memory. However, this operation is off the critical path as the fetching occurs while the TBs are executing on the SM. The only time the penalties occur is when there is a free TB context available in an SM but there is no ready TB to be issued. However, this scenario is very rare as the time taken for loading a TB from memory is very small compared to the TB execution time.

5.2 Task Stealing

Towards the end of the execution run, if the scheduling of a group to an SM leads to load imbalance, we utilize task stealing to balance out the load of the last group amongst SMs. Note that since PAVER focuses on RAR locality, rather than RAW, it will not preclude cases where issued thread blocks on a busy GPU are dependent on still-unissued TBs which can result in a deadlock.

When there are no more unassigned groups, we employ *task stealing* to improve performance through load-balancing. Ideally SMs should finish their workloads all at the same time. However, there are workloads of different sizes on each SM. Therefore, an SM could finish early and stay idle while the other SMs are still working on their TB groups. With task stealing, however, we take a TB assigned to a busy SM (but not yet issued) and reassign it to a free SM. By tapping into the freed resources, we can make sure that the SMs are utilized as much as possible until the application's termination, resulting in additional performance.

Task stealing is employed in k-way-TS and RB-TS. In MST-TS, the TB-Group size is 1, hence a SM does not have more than 1 TB waiting in the TB group. When all the tasks in the TB group are exhausted and SM has a free TB context, it is marked as recipient SM. While, the SM with maximum number of waiting tasks in the TB group is the donor SM. The "WaitingTB" of each SM is determined by the number of waiting tasks in its TB group (obtained from $SM.next$ and $SM.tail$). The granularity of tasks stolen is determined as: $Max_{WaitingTB}$ - $Average_{WaitingTB}$.

Algorithm 2: Task Stealing Algorithm

```
1  Max_WaitingTB=0
2  Average_WaitingTB=0
3  for SM in range(0, total SM) do
4      WaitingTB= SM.tail - SM.next
5      if WaitingTB > Max_WaitingTB then
6          Max_WaitingTB=WaitingTB
7          DonorSM=SM
8      end
9      Average_WaitingTB += WaitingTB
10 end
11 Average_WaitingTB /= total SM
12 Stolen TB count = Max_WaitingTB - Average_WaitingTB
13 return DonorSM, Stolen TB count
```

If $Stolen\ TB\ count > 2$, it still captures some locality. Task stealing affects the L1 locality within the donor SM but the overall kernel execution saves the extra cycles by load-balancing the SMs. Also, the locality at L2 level is still preserved.

Storage Overhead: For the task-stealer, 3 16-bit registers are used to store the $Max_{WaitingTB}$, $Average_{WaitingTB}$, *Stolen TB count* which accounts for 6 Bytes. This overhead is negligible compared with the area of other on-chip storage structures in Fermi, Pascal and Volta GPU.

Control logic Overhead: The task stealer uses a very simple control circuit consisting of 3 adders, 1 comparator and 1 divider. The area and power overhead of these control logic would be negligible compared to the GPU chip area.

5.3 Generalized Runtime Algorithm for all TB Policies

Once the kernel starts executing, TB scheduler assigns the TB groups from the global queue to the SMs in a round-robin manner. As shown in Figure 9, once a TB group is assigned to an SM, the $SM.next$ and $SM.tail$ are initialized to point to the TB group's head and tail, respectively. Due to limited hardware resources, a limited number of TBs can be issued and executed concurrently in an SM. A TB (stored in *SM.nextTB*) from the *assigned TB group* is *issued* to the SM as soon as a free TB context is available. Upon TB issue, SM.next is updated to point to next TB in the assigned TB-Group. TB-scheduler then fetches the *nextTB* to be issued, located at TB Group[SM.next] and fills up the *SM.nextTB* register. If $SM.next == SM.tail$, then all the TBs in the TB group are exhausted. The PAVER TB scheduler will then assigns another available TB group to the SM.

When all the entries in the global queue are exhausted, the load imbalance occurs, as an SM has exhausted its TB group and has a free TB context available, while other busy SMs have TBs waiting in their assigned TB groups. In this scenario, the task stealing is enabled. The donor SM and number of TBs to steal are determined as per the task stealing algorithm (described in Algorithm 2). The tasks are stolen from the tail of the donor TB group and the tail of the donor SM is updated. After the task stealing, $SM.next$ and $SM.tail$ are updated accordingly. For example: There are 4 waiting TBs in the TB-Group of the donor SM indexed as (TB_0, TB_1, TB_2, TB_3), $DonorSM.next = TB_0$, $DonorSM.tail = TB_3$ and $Stolen\ TB\ count = 2$. After Task-stealing, recipient $SM.tail$ is updated to donor SM.tail i.e. TB_3. Donor $SM.tail$ is decremented by the stolen TB count and points to TB_1. Recipient SM.tail points to $(DonorSM.tail + 1)$ i.e. TB_2.

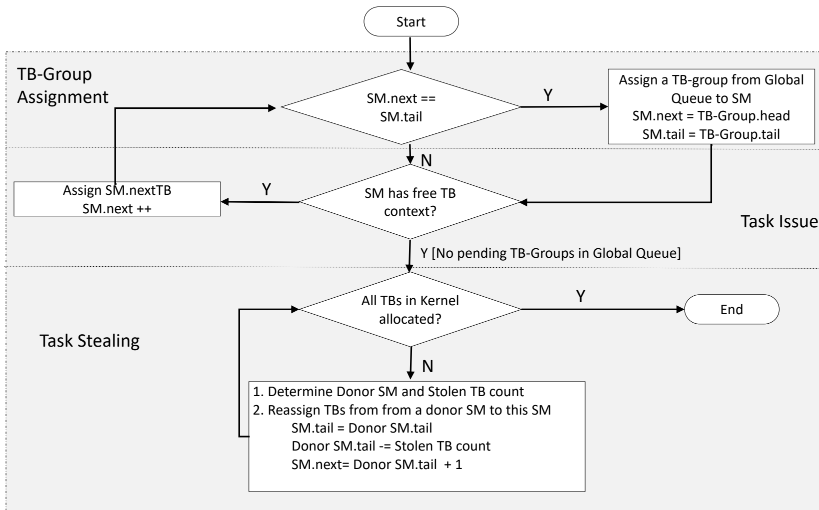

Fig. 9. PAVER Runtime Flowchart

6 EVALUATION

6.1 Methodology

We use GPGPU-Sim [5] with simulation parameters in Table 1 to model GTX480 Fermi, TITANX Pascal and TITANV Volta GPUs. The warp scheduling policy follows a greedy-then-oldest (GTO) policy [48]. Our thread block scheduling technique can be run with any warp scheduler, but we find GTO to provide the best performance. Apart from our MST-TS, k-way-TS and RB-TS, we also implement the Loose Round Robin (LRR) and Block CTA Scheduler (BCS) scheduling policy as the baseline. We did not compare with warp scheduling policies, such as CCWS [49], because they are orthogonal to TB scheduling and can be applied to PAVER to further improve the performance.

LRR scheduler selects one SM at a time, and assigns a TB in a sequential order. LRR is one of the the fairest schedulers. However, since non-adjacent TBs are assigned to the same SM, there is less locality among them and their data accesses may cause the L1 cache to suffer from thrashing. Meanwhile, the locality will mostly remain on the L2 cache level because adjacent TBs will execute at the same time but in different SMs. BCS [29] is similar to LRR with one difference: it schedules two neighboring TBs at the same time with the assumption that the highest locality among TB occurs in neighboring pairs. This specifically benefits 2D grid-type workloads only. Since BCS will only schedule if the SM has enough resources to fit in two new TBs, it can potentially lead to performance reduction due to lack of resources. In the worst case, if no free context for a TB pair can be found during the execution, it may add unnecessary cycles to the execution by causing task serialization, whereas it could avoid such cases by filling any free context in the SM without pairing the remaining TBs in that cycle. BCS has better L1 locality than LRR, but it can be improved much more. The L2 locality remains the same as LRR.

6.2 Benchmarks

Benchmarks shown in Table 2 are selected from Parboil [54], Rodinia [7], Polybench [45], ISPASS [1], and CUDA SDK [36] benchmark suites. All these suites are widely used in evaluating a GPU architecture's performance. Out of all the kernels in these suites, we used a subset that have high as well as low inter-TB locality. However, we also used a few benchmarks with little to no locality

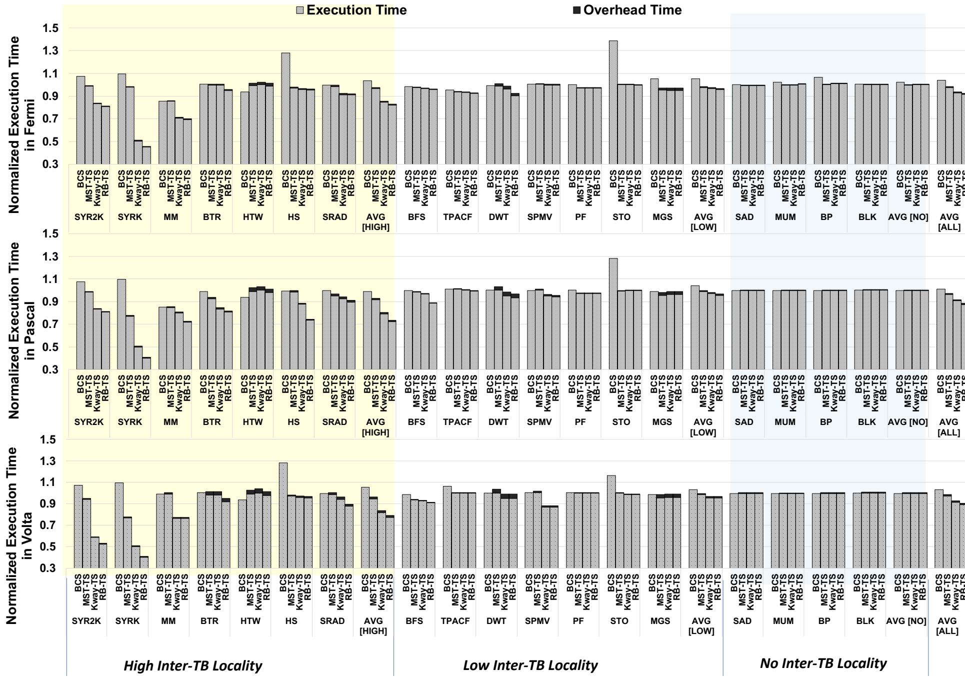

Fig. 10. Kernel Execution Time and JIT Analysis Overhead of BCS, MST-TS, k-way-TS, RB-TS normalized w.r.t. baseline TB scheduling policy (LRR), on Fermi (top row), Pascal (middle) and Volta architectures (bottom).

to test the impact of our TB scheduling policies to see if they are affected in any way. As shown in Figure 2, The benchmarks such as SYR2K, SYRK,MM, BTR, HTW, HS and SRAD which have at least 50% Inter TB Data-references (Inter-TB + Intra-TB ∩ Inter-TB) are considered high-TB locality. The benchmarks with less than 50% Inter TB Data-references are considered low-TB locality. The benchmarks which show low-TB locality are BFS, TPACF, DWT, SPMV, PF, STO and MGS. The benchmarks without any inter-TB locality are SAD, MUM, BP and BLK.

6.3 Results

Speedup: Figure 10 displays the kernel execution time as well as JIT Analysis Overhead of our different TB scheduling policies with respect to LRR and BCS. The x-axis shows the benchmark name, and the y-axis shows the Execution time + JIT Overhead w.r.t. LRR. The results have been normalized to the LRR TB scheduler. The TB policies LRR and BCS do not involve JIT analysis, hence, JIT time is excluded in BCS result. GPU kernel execution time is calculated using the kernel execution cycles and core frequency (from Table 1). On an average, the JIT overhead is observed to be a very negligible fraction (1%) of the total kernel execution time.

Figure 11 displays the speedup of our different TB scheduling policies with respect to LRR and BCS. The x-axis shows the benchmark name, and the y-axis shows the speedup w.r.t. LRR. The results have been normalized to the LRR TB scheduler. The speedup of PAVER TB scheduling policies over the baseline (LRR) is calculated as $Speedup = \frac{kernel\ execution\ time\ in\ baseline}{kernel\ execution\ time\ in\ PAVER{+}JIT\ analysis\ time}$. In Fermi (Figure 11 (top)), for applications with high inter-TB locality, on an average, the TB policies: Maximum Spanning Tree-based TB Scheduler (MST-TS), k-way partition based TB Scheduler (k-way-TS) and Recursive Bipartition-based TB Scheduler (RB-TS) have 2.8%, 23.8% and 29% speedups

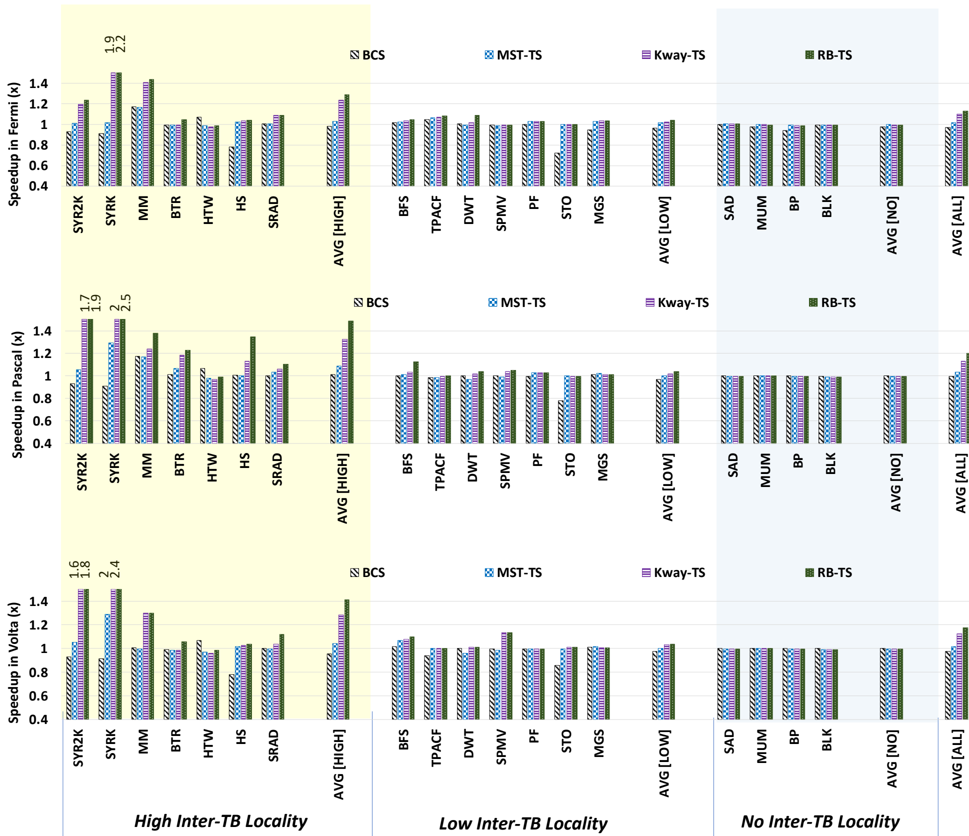

Fig. 11. Speedup of BCS, MST-TS, k-way-TS, RB-TS normalized w.r.t. baseline TB scheduling policy (LRR), on Fermi (top row), Pascal (middle) and Volta architectures (bottom).

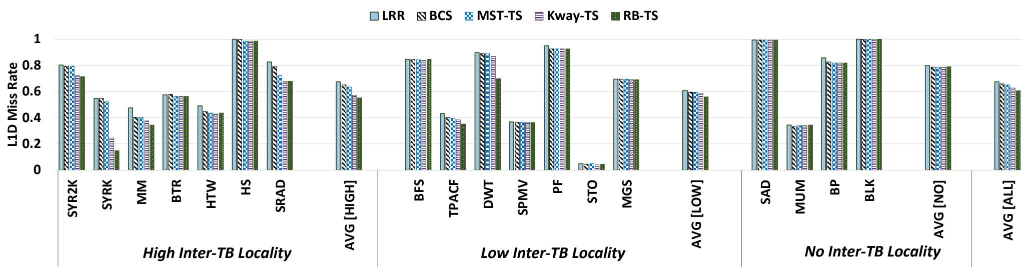

Fig. 12. L1 miss rate comparison of LRR, BCS, MST-TS, k-way-TS and RB-TS in Fermi.

respectively as compared to baseline LRR. The average speedup for applications with low inter-TB locality are for MST-TS, k-way-TS and RB-TS are 1.8%, 2.5% and 3.8% respectively. We also tested our TB policies on the benchmarks SAD, MUM, BP and BLK which do-not have any inter-TB locality to see if the IPC is affected by the new TB scheduling policies. It is observed that the performance of our proposed TB policies for these applications is reduced by a negligible amount 0.15%,0.3% and 0.5% for MST-TS, k-way-TS and RB-TS respectively as compared to the baseline LRR scheduler.

Overall, considering the applications from different inter-TB locality categories (high, low and no-sharing), MST-TS, k-way-TS and RB-TS show an average speed-up of 1.8%, 10.1% and 12.6% respectively. PAVER was evaluated for the recent architectures Pascal and Volta. Our graph-based TB policies MST-TS, k-way-TS and RB-TS achieve an average speedup of 10.8%, 32.5% and 49.1% for high inter-TB benchmarks; 0.2%, 1.8% and 3.7% for low inter-TB benchmarks; and 3.3%, 13.32% and 20.49% overall for different inter-TB locality applications in Pascal (Figure 11 (middle)). We observe that PAVER fares well in Volta (Figure 11 (bottom)), where MST-TS, k-way-TS and RB-TS achieve an average speedup of 4.5%, 28.6% and 41.2% for high inter-TB benchmarks; 0.4%, 3.5% and 3.8% for low inter-TB benchmarks; and 0.18%, 12.4% and 17.4% overall for different inter-TB locality applications.

The high inter-TB locality benchmarks get significant improvement in IPC through our graph-based TB scheduling policies, upto 2.2x for SYRK benchmark (Fermi). Note that our work is orthogonal to warp scheduling techniques, and therefore adding a warp scheduler [18, 35] on top of the TB scheduler could further increase the speedup.

In the case of BCS, if the maximum thread block (TB) per SM limit is an odd number, it leads to thread block throttling, which means some of the thread blocks could not be jointly assigned due to the lack of free TB contexts in the SM. For example, in the STO application, there are 384 thread blocks and yet 3 thread blocks executing on the SM at the same time (maximum concurrent TB execution per SM depends on the resources like registers, local memory, shared memory, constant memory consumed by each TB). The BCS suffers drastically because every SM is assigned to execute 2 thread blocks instead of 3 as in the baseline. Similarly, in HS benchmark 3 TBs execute concurrently in a SM and lead to severe throttling incase of BCS TB policy which leads to around 22% slower execution than LRR. Similarly, for benchmarks like SYR2K, SYRK, BP and MGS even if the maximum TB per SM is even i.e. 6, 6 ,6 and 8 respectively, whenever a TB finishes execution in a SM, BCS does not start executing the next TB immediately. Rather, it waits for 2 TB contexts to be freed up so that a pair of TB can start executing. This leads to SM resources starvation and inferior performance by increasing the execution time of SYR2K, SYRK, BP and MGS by 7%, 9% ,6% and 5% respectively w.r.t LRR. Our policies, however, do not lead to throttling as evident in Figure 11. In addition, BCS assigns every two consecutive TBs to the SM based on the assumption that the neighboring TBs would always have a high inter-TB locality, which does not hold for 2- and higher-dimensional grid applications, where the locality could be between TBs in a column. Our graph-based approaches, however, are generalized methods and take all types of data access patterns into account.

It may be pointed out that prior work [31] covers a limited pattern behavior which has locality along the X-dimension or Y-dimension of the grid. In our paper, any locality pattern (shown in Figure 5) can be analyzed in form of a graph and partitioned among the SMs to leverage maximum locality within the SM. Applications such as BTR, BFS and SPMV having irregular data-locality pattern application have been analyzed through our graph-based approach and gives significant speedup for the Fermi, Pascal and Volta architectures as shown in Figure 11. We get significant performance benefits compared to the baseline LRR for the unstructured applications BTR (4.52% for Fermi, 22.8% for Pascal and 5.9% for Volta), BFS (4.4% for Fermi, 12.8% for Pascal and 9.85% for Volta); and SPMV (-0.5% for Fermi, 4.9% for Pascal and 13.7% for Volta).

Effect of Task Stealing: When all the thread blocks on an SM finish early and the SM is idle, the thread blocks on a busy SM are reassigned to the free SM for the load balancing purpose. Task stealing is beneficial for only k-way-TS and RB-TS approach as in these approach the TB groups are pre-assigned to the SM without accounting for the actual execution time of each SM. How-ever in the MST-TS the load-balancing is done implicitly when the TB from the MST are assigned to the SM in a round-robin fashion. Task stealing re-balances the workload of each SM when the kernel

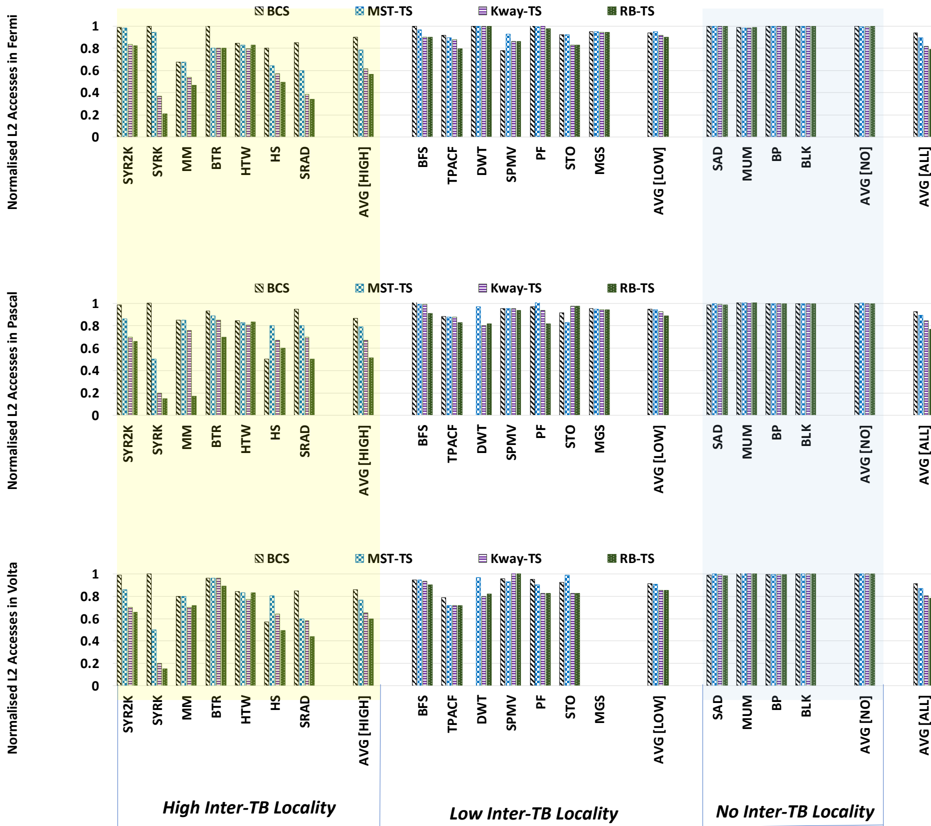

Fig. 13. L2D access comparison for BCS, MST-TS, k-way-TS and RB-TS normalized w.r.t. LRR for applications with high, low and no inter-TB locality, on Fermi (top), Pascal (middle) and Volta architectures (bottom).

is on the verge of finishing the execution and some of the SMs have already finished their work. Average Performance benefits of 3% and 2% were observed for k-way-TS and RB-TS respectively w.r.t no task stealing case in Fermi.

Effect of Cache Size: Increasing the cache size shall reduce the capacity misses in the cache. However, the cache in GPUs tend to be limited in size. Since in the Fermi architecture 64KB of RAM had configurable partitioning between shared memory and L1 Cache, we increased the L1 Cache size from 16 KB to 48 KB to see the effect of the cache size on the performance of applications. With increased cache size PAVER[RB-TS] outperforms the baseline configuration by 16%.

Miss Rates: Figure 12 shows the L1 miss rate of different TB scheduling methods explored in this paper; MST-TS, k-way-TS, RB-TS, BCS and LRR. It can be seen that our recursive bipartitioning method reduces the L1 miss rate over LRR by 43.3% (high inter TB locality), 10% (low inter TB locality) and 21% (all applications), contributing to the speedup. The L1 miss rate shown in Figure 12 is proportional to the L2 accesses shown in Figure 13 (Fermi). Hence, we excluded the L1 miss rate results for Pascal and Volta as normalised L2 accesses accounts for that.

The average conflict and capacity miss for all the benchmarks is 9% and the cold miss constitutes 91% of the total miss at L2. Hence, we observe very minimal reduction in L2 misses as majority

of them are cold misses, however the accesses to L2 are reduced by 21% as the graph-based TB policies reduce the misses at L1. Reduced accesses to L2 cache and shorter execution time leads to energy saving. Figure 13 shows the L2 accesses for all the TB scheduling policies normalized to baseline LRR. In Fermi (Figure 13(top), TB policies MST-TS, k-way-TS and RB-TS lead to reduced L2 accesses of 78.4%, 61.4% and 56.7% for high inter-TB benchmarks; 95%, 91.88% and 90.22% for low inter-TB benchmarks; and 89.6%, 81.7% and 79.3% overall for all applications. In Pascal (Figure 13(middle)), TB policies MST-TS, k-way-TS and RB-TS lead to reduced L2 accesses of 79%, 67% and 51.59% for high inter-TB benchmarks; 94.2%, 92.6% and 89.1% for low inter-TB benchmarks; and 89.6%, 84.3% and 76.94% overall for different inter-TB locality applications. L2 transactions are reduced for Volta (Figure 13 (bottom)) , where MST-TS, k-way-TS and RB-TS reduce accesses by significant fraction of 76.4%, 65% and 59.8% for high inter-TB benchmarks; 90.9%, 85.5% and 85.3% for low inter-TB benchmarks; and 87%, 80.5% and 78.3% overall for all applications.

It is noteworthy that in both k-way partitioning (k-way-TS) and recursive bipartitioning (RB-TS), in cases where the number of TBs are less than the total SM capacity, i.e. total TBs that all SMs can hold, all thread blocks are assigned to their respective SMs immediately after the kernel's initialization. This means that both schedulers would perform the same.

Locality captured in each partition is compared k-way-TS and RB-TS. Locality is expressed in-terms of the sum of the edge weights of the sub-graph in the partition. The more is the edge weight of each partition, more is the locality captured. In k-way the partition size is huge and the concurrently running TB might not necessarily have the maximum sharing within the partition. How-ever in RB-TS we reach a sweet-spot where all the concurrently running TB i.e. all the TB inside a partition are likely to have maximum sharing. This is why we observe lower L1 miss rates in RB-TS as compared to k-way-TS.

Comparison with cache-fit graph partitioning policy : A prior work [9] has employed cache-fit policy on SPMV application using edge-partitioning and kernel-splitting. To perform a comparison, we evaluated SPMV application in Parboil benchmark suite for cache-fit policy using GPGPU-Sim. Through benchmarking of the SPMV execution, the TB size was found out to be 15 KB. Since the default L1 size in Fermi is 16KB, 1 TB is mapped to the SM in the cache-fit policy. In our policy, we had employed 5 TBs based on the register and threads usage per TB. Similarly, the number of TBs per SM was reduced to 3 and 2 for Pascal and Volta architectures so that the working set of the concurrent TBs fits into the L1 cache. Overall, we observed that the normalized speedup is reduced in the cache-fit policy compared to the baseline LRR policy. They are 58% (Fermi), 91.24% (Volta) of the baseline. Reducing the number of concurrent TBs in the SM in cache-fit policy helps reduce the pressure on L1 cache, resulting in lower L1 cache misses. However, it also results in the under utilization of a SM and starvation of the other SM resources resulting in increased execution time. In Pascal, the number of TBs mapped to SM is set to 3 due to a larger L1 size, resulting in a speed-up of 109.73% over baseline. However, it may be reminded that the RB-TS policy in PAVER is even better than the baseline LRR by 99.4% (Fermi), 104.94% (Pascal) and 114.9% (Volta). In cache-fit policy, the normalised L1 miss rate is reduced compared to the baseline, 42.72% (Fermi), 87.32% (Pascal), 100.6% (Volta) of the baseline, similarly, in RB-TS, the normalized L1 miss rate is reduced compared to the baseline, 86.18% (Fermi), 94% (Pascal), 99% (Volta) of the baseline. However, the L1 miss-rates are reduced in our RB-TS for all architectures without under-utilizing the other SM resources, resulting in a high speedup.

7 RELATED WORK

Cache Locality: Koo et al [25] categorize the load instructions into deterministic, where the address is calculated using thread ID, thread block ID, etc., and non-deterministic (from user input, etc.) Deterministic loads are observed to have a more coalesced access pattern in a stark contrast to

the non-deterministic loads which can create far more reservation fails in the cache. They then suggest solutions to alleviate the issue with such loads, such as prefetching for indirect addresses [27], reworking the cache hierarchy, and assigning neighboring TBs with data locality to the same SM, the last one being the focus of our work. Vijaykumar et al [59] has also shown the potential of exploiting locality by proposing the 'Locality Descriptor', which enables definition of locality abstractions on software by the programmer, and utilization thereof by the hardware, improving performance when exploiting locality in the caches. Our work lifts the requirement of software abstraction definition from the programmer, and instead seeks to use a generalized compiler-based approach to extract the block locality among the TBs and utilize them when assigning SMs.

TB Scheduling: Making use of the locality among thread blocks can prove challenging since very little is known about the exact underlying TB scheduling architecture. Several cache locality algorithms and structures have been proposed for GPGPUs in recent years which include some form of a thread block scheduler within. As mentioned earlier, most of them have a rather naive approach as they develop heuristics based on workload behavior (e.g. data layout) to exploit data locality. Another drawback is that many of them target specific structures only, e.g. grid applications [57]. Our work, however, focuses on a more generalized approach to exploit data locality where there is no need to know the application's access patterns, making it effective for applications with all types of structures.

Chen et al [8] propose a hardware-software approach for applications with structural data access, both row- and column-major applications. It checks the address ranges of the ready TBs and issues the TB with the maximum overlapping address range with the TBs already executing on that SM, increasing data reuse and improving the performance. The maximum overlapping address is determined with the assumption that each TB accesses a continuous 2D space in the cache, whereas our work has a more generalized approach.

Also, in [55], Abdulaziz et al. devise a sharing-aware TB scheduler. Based on their observation that around 70% of data sharing takes place between consecutive thread block IDs. It assigns TB groups of consecutive TB IDs to the SMs while maintaining load balance among SMs. To aid the scheduler, they also devise a cache replacement policy in L1 and L2 levels which prevents cache block duplication in L1 and L2. Most of their performance benefits come from efficient cache replacement policies as compared to their TB scheduling policy. The idea of temporal locality could also be extended to dynamic parallelism [19]. Wang et al [60] propose a locality-aware scheduler specifically for dynamic parallelism, in which the TBs belonging to child kernel will be scheduled on the same SM as those of the parent kernel are on.This paper shows that similar to parallel TBs, locality among parent and child TBs can also be high and therefore exploited.

Graph Partitioning based on Resource: In some of the emerging architectures, graph partitioning algorithms can be used to assign tasks to the best execution unit available. In [44], a wafer-scale architecture is proposed to minimize communication overheads and memory access latency. It aims to schedules thread blocks with high data sharing to adjacent processing modules. The partitioning algorithm used seeks to minimize the number of edges moving across the partitions, i.e. shared data references. However, for non-neighboring devices to share data, multiple "hops" are required among the modules, which can dramatically reduce the performance.

Warp Scheduling: Augmenting the warp schedulers exists in many works for different applications, including exploitation of data locality within the TB. In [49], the warp scheduler rearranges the access patterns of different warps as well as the threads in the warps to reduce the L1 cache misses and thrashing significantly, resulting in 24% performance improvement. The same authors proposed [50] which checks the control flow of the execution and divergences, and uses a predictive approach to schedule threads such that the L1 cache size usage and the likelihood of thrashing is minimized, resulting in a 26% improvement over [49].

Oh et al [43] propose a locality-aware warp scheduler coupled with a prefetching scheme, yielding 31.7% performance improvement over the baseline. In this work, cache access patterns are analyzed and warps accessing the same cache line in the same time frame are grouped. If the first warp of the group hits the cache, the rest of the group will also be prioritized under the assumption that their accesses would also be hits, increasing data utilization before eviction. Should the first warp miss, the prefetcher would attempt to pre-load the data for the other warps in the group as well.

It must be pointed out that we consider only spatial locality among the TBs because the temporal localities depend on the warp scheduling policies inside a streaming multiprocessor (SM). These are best handled by warp scheduling policies, such as CCWS [49], which can work orthogonal to the TB scheduling policies. When applied with PAVER, they will further improve the performance.

Locality in CPU-related works: The idea of using data locality to improve the performance and reduce the memory bottleneck started in the realm of CPUs and continued to multicores. As the cores have become more complex, so have the cache hierarchies, which means that any unnecessary cache action can now leave more of a negative impact on the performance and the energy efficiency [61]. According to [64], even though exploiting inter-core cache locality is in progress, it should not be without taking intra-core locality into account, for it could actually perform worse than only exploiting intra-core locality. There have been analyses of the tradeoff between cache reuse and vectorization on CPUs, but in the end, they should be used in the right place and it mainly depends on the application type and the architecture [52]. Kandemir et al [20] argue that different cache hierarchies in different architectures makes it difficult for the programmer to optimize the application for all architectures, and presents a compiler-based method in which loop iterations are assigned to different cores and scheduled based on the cache topology and cache access pattern.

In [64], Zhang et al. use a compiler-based strategy to create a computation block dependency graph, targeting data reuse on multicore CPUs. Their observation on data reuse balance leads them to develop a task mapping and scheduling policy that balances inter-core and intra-core data reuse.

Task stealing has also been incorporated and explored in recent literature. Yoo et al [61] propose a locality-aware scheduler for unstructured parallel applications in a multi-core CPU which increases the speedup and reduces the energy consumption for a 32-core system by 2.05x and 47% respectively, and shows that the benefit will increase as the number of cores increase, an additional 1.83x for 1024 cores. They capture the data sharing of an application using special programming APIs and use this information to create a task sharing graph. Then, they generate task groups to be launched on each core keeping the cache topology in consideration. They also perform task reordering to capture temporal locality and task stealing for load balancing.

The work proposed by Lifflander et al [33] increases cache locality for recursive programs by tracking data reuse opportunities and, using work stealing, interleaving the execution of the function that can use them.

8 CONCLUSION

In this paper, we have shown that the inter-TB locality can be exploited to improve the performance substantially. Unstructured parallelism is a part of many GPU applications today and newer architectures should take advantage of the data locality among the thread blocks. To this end, we performed compiler analysis to measure the data reference sharing among TBs for various applications. Then we proposed three generalized graph-based TB scheduling policies based on MST, k-way partitioning, and recursive bipartitioning. Our scheduling techniques reduce L2 accesses by 43.3%, 48.5%, 40.21% and increase the average performance speedup by 29%, 49.1% and 41.2% . We believe the results can be further improved by taking the time of data reference sharing into consideration.

ACKNOWLEDGMENT

We would like to thank Dr. V. Krishna Nandivada for identifying the locality information for the locality graph generation problem. This work is partly supported by NSF Grants CCF-1815643 and CCF-1907401.

REFERENCES

[1] 2009. https://github.com/gpgpu-sim/ispass2009-benchmarks. Accessed: 2018-04-11.

[2] AmirAli Abdolrashidi, Devashree Tripathy, Mehmet E Belviranli, Laxmi N Bhuyan, and Daniel Wong. 2017. Wireframe: Supporting Data-dependent Parallelism through Dependency Graph Execution in GPUs. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 50)*. 600–611.

[3] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. 2000. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1–12.

[4] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[5] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 163–174.

[6] Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Laxmi N Bhuyan. 2018. Juggler: a dependence-aware task-based execution framework for GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 54–67.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 44–54.

[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 Computer Architecture Letters* 16, 2 (2017), 127–131.

[9] Yanhao Chen, Ari B Hayes, Chi Zhang, Timothy Salmon, and Eddy Z Zhang. 2018. Locality-aware software throttling for sparse matrix operation on GPUs. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 413–426.

[10] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 353–364.

[11] Hodjat Asghari Esfeden, Amirali Abdolrashidi, Shafiur Rahman, Daniel Wong, and Nael Abu-Ghazaleh. 2020. BOW: Breathing operand windows to exploit bypassing in GPUs. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 996–1008.

[12] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. 2006. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 83.

[13] Naznin Fauzia, Louis-Noël Pouchet, and P Sadayappan. 2015. Characterizing and enhancing global memory data coalescing on GPUs. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 12–22.

[14] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. 2010. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *ACM Sigplan Notices*, Vol. 45. ACM, 341–342.

[15] Bruce Hendrickson and Robert Leland. 1993. *The Chaco users guide. Version 1.0.* Technical Report. Sandia National Labs., Albuquerque, NM (United States).

[16] Muhammad Huzaifa, Johnathan Alsop, Abdulrahman Mahmoud, Giordano Salvador, Matthew D Sinclair, and Sarita V Adve. 2020. Inter-kernel Reuse-aware Thread Block Scheduling. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 3 (2020), 1–27.

[17] Ali Jahanshahi, Hadi Zamani Sabzi, Chester Lau, and Daniel Wong. 2020. GPU-NEST: Characterizing Energy Efficiency of Multi-GPU Inference Servers. *IEEE Computer Architecture Letters* 19, 2 (2020), 139–142.

[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 *ACM SIGPLAN Notices*, Vol. 48. ACM, 395–406.

[19] Stephen Jones. 2012. Introduction to dynamic parallelism. In *GPU Technology Conference Presentation S*, Vol. 338. 2012.

[20] Mahmut Kandemir, Taylan Yemliha, SaiPrashanth Muralidhara, Shekhar Srikantaiah, Mary Jane Irwin, and Yuanrui Zhnag. 2010. Cache topology aware computation mapping for multicores. In *ACM Sigplan Notices*, Vol. 45. ACM, 74–85.

[21] George Karypis and Vipin Kumar. 1998. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN* (1998).

[22] Mahmoud Khairy, Vadim Nikiforov, David Nellans, and Timothy G Rogers. 2020. Locality-Centric Data and Threadblock Management for Massive GPUs. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1022–1036.

[23] Farzad Khorasani, Hodjat Asghari Esfeden, Amin Farmahini-Farahani, Nuwan Jayasena, and Vivek Sarkar. 2018. Regmutex: Inter-warp gpu register time-sharing. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 816–828.

[24] Hyunjun Kim, Sungin Hong, Hyeonsu Lee, Euiseong Seo, and Hwansoo Han. 2019. Compiler-Assisted GPU Thread Throttling for Reduced Cache Contention. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[25] Gunjae Koo, Hyeran Jeon, and Murali Annavaram. 2015. Revealing critical loads and hidden data locality in gpgpu applications. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 120–129.

[26] Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.

[27] Nagesh B Lakshminarayana and Hyesoon Kim. 2014. Spare register aware prefetching for graph algorithms on GPUs. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 614–625.

[28] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[29] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 260–271.

[30] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. 2009. Fermi architecture white paper.

[31] 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 Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 297–311.

[32] Yun Liang, Xiaolong Xie, Yu Wang, Guangyu Sun, and Tao Wang. 2017. Optimizing cache bypassing and warp scheduling for GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 8 (2017), 1560–1573.

[33] Jonathan Lifflander and Sriram Krishnamoorthy. 2017. Cache locality optimization for recursive programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1–16.

[34] Hoda Naghibijouybari, Khaled N Khasawneh, and Nael Abu-Ghazaleh. 2017. Constructing and characterizing covert channels on GPGPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 354–366.

[35] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 308–317.

[36] NVIDIA. 2007. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. Accessed: 2018-04-11.

[37] NVIDIA. 2009. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[38] NVIDIA. 2016. GeForce GTX 1080. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.

[39] NVIDIA. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed: 2018-11-26.

[40] NVIDIA. 2020. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#just-in-time-compilation. Accessed: 2020-9-23.

[41] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf. Accessed: 2020-10-08.

[42] CUDA NVIDIA. 2012. C Programming Guide, v4.2, April 2012.

[43] Yunho Oh, Keunsoo Kim, Myung Kuk Yoon, Jong Hyun Park, Yongjun Park, Won Woo Ro, and Murali Annavaram. 2016. APRES: improving cache efficiency by exploiting load characteristics on GPUs. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 191–203.

[44] Saptadeep Pal, Daniel Petrisko, Matthew Tomei, Puneet Gupta, Subramanian S Iyer, and Rakesh Kumar. 2019. Architecting Waferscale Processors-A GPU Case Study. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 250–263.

[45] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* (2012).
[46] Robert Clay Prim. 1957. Shortest connection networks and some generalizations. *Bell Labs Technical Journal* 36, 6 (1957), 1389–1401.
[47] K. Ranganath, A. Abdolrashidi, S. L. Song, and D. Wong. 2019. Speeding up Collective Communications Through Inter-GPU Re-Routing. *IEEE Computer Architecture Letters* 18, 2 (2019), 128–131. https://doi.org/10.1109/LCA.2019.2933842
[48] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 72–83.
[49] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 72–83.
[50] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2013. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 99–110.
[51] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. 2005. Cache aware optimization of stream programs. *ACM SIGPLAN Notices* 40, 7 (2005), 115–126.
[52] Du Shen, Milind Chabbi, and Xu Liu. 2018. An Evaluation of Vectorization and Cache Reuse Tradeoffs on Modern CPUs. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 21–30.
[53] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. 2018. CUDAAdvisor: LLVM-based runtime profiling for modern GPUs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 214–227.
[54] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).
[55] Abdulaziz Tabbakh, Murali Annavaram, and Xuehai Qian. 2017. Power Efficient Sharing-Aware GPU Data Management. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 698–707.
[56] Devashree Tripathy, Hadi Zamani, Debiprasanna Sahoo, Laxmi N Bhuyan, and Manoranjan Satpathy. 2020. Slumber: static-power management for gpgpu register files. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 109–114.
[57] S. Tripathy, D. Sahoo, and M. Satpathy. 2019. Multidimensional Grid Aware Address Prediction for GPGPU. In *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*. 263–268. https://doi.org/10.1109/VLSID.2019.00064
[58] Dominic A Varley. 1993. Practical experience of the limitations of gprof. *Software: Practice and Experience* 23, 4 (1993), 461–463.
[59] 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. ISCA.
[60] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2016. LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 583–595.
[61] Richard M Yoo, Christopher J Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. 2013. Locality-aware task management for unstructured parallelism: A quantitative limit study. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. 315–325.
[62] Hadi Zamani, Yuanlai Liu, Devashree Tripathy, Laxmi N. Bhuyan, and Zizhong Chen. 2019. GreenMM: energy efficient GPU matrix multiplication through undervolting. In *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*. 308–318. https://doi.org/10.1145/3330345.3330373
[63] Hadi Zamani, Devashree Tripathy, Laxmi Bhuyan, and Zizhong Chen. 2020. SAOU: safe adaptive overclocking and undervolting for energy-efficient GPU computing. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 205–210.
[64] Yuanrui Zhang, Mahmut Kandemir, and Taylan Yemliha. 2011. Studying inter-core data reuse in multicores. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. ACM, 25–36.
[65] Intel Developer Zone. 2017. Intel VTune Amplifier, 2017. *Documentation at the URL: https://software. intel. com/en-us/intel-vtune-amplifier-xe-support/documentation* (2017).