Locality-Aware CTA Clustering for Modern GPUs

Ang Li
Pacific Northwest National Lab
ang.li@pnnl.gov

Shuaiwen Leon Song
Pacific Northwest National Lab
shuaiwen.song@pnnl.gov

Weifeng Liu
University of Copenhagen
weifeng.liu@nbi.ku.dk

Xu Liu
College of William and Mary
xl10@cs.wm.edu

Akash Kumar
Technische Universität Dresden
akash.kumar@tu-dresden.de

Henk Corporaal
Eindhoven University of Technology
h.corporaal@tue.nl

Abstract

Cache is designed to exploit locality; however, the role of on-chip L1 data caches on modern GPUs is often awkward. The locality among global memory requests from different SMs (Streaming Multiprocessors) is predominantly harvested by the commonly-shared L2 with long access latency; while the in-core locality, which is crucial for performance delivery, is handled explicitly by user-controlled scratchpad memory. In this work, we disclose another type of data locality that has been long ignored but with performance boosting potential — the inter-CTA locality. Exploiting such locality is rather challenging due to unclear hardware feasibility, unknown and inaccessible underlying CTA scheduler, and small in-core cache capacity. To address these issues, we first conduct a thorough empirical exploration on various modern GPUs and demonstrate that inter-CTA locality can be harvested, both spatially and temporally, on L1 or L1/Tex unified cache. Through further quantification process, we prove the significance and commonality of such locality among GPU applications, and discuss whether such reuse is exploitable. By leveraging these insights, we propose the concept of CTA-Clustering and its associated software-based techniques to reshape the default CTA scheduling in order to group the CTAs with potential reuse together on the same SM. Our techniques require no hardware modification and can be directly deployed on existing GPUs. In addition, we incorporate these techniques into an integrated framework for automatic inter-CTA locality optimization. We evaluate our techniques using a wide range of popular GPU applications on all modern generations of NVIDIA GPU architectures. The results show that our proposed techniques significantly improve cache performance through reducing L2 cache transactions by 55%, 65%, 29%, 28% on average for Fermi, Kepler, Maxwell and Pascal, respectively, leading to an average of 1.46x, 1.48x, 1.45x, 1.41x (up to 3.8x, 3.6x, 3.1x, 3.3x) performance speedups for applications with algorithm-related inter-CTA reuse.

CCS Concepts • **Computer systems organization → Single instruction, multiple data**; • **Software and its engineering → Runtime environments**

Keywords GPU, CTA, cache locality, performance optimization, runtime tool

1. Introduction

Loop tiling, also known as *loop blocking* or *strip-mine-and-interchange*, has long been proven to be effective on optimizing loop-nests in CPU programs [1–3]. Loop tiling partitions the loop iteration space into smaller chunks so that reusable data referenced within the tile can be kept in the on-chip cache, avoiding early eviction. Listing 1 shows an example of matrix-vector multiplication. In the original loop nest (Loop1), *C[i]* has reuse across loop j while *B[j]* has reuse across loop i. *A[i][j]* is a streaming access and has no reuse. After tiling both i and j, Loop2 can be obtained. The intention of titling is to reduce the data set (from $n \times n$ iteration space to $m \times m$ tile space) so that *C[x]* and *B[y]* can be reused within the tile and kept in the on-chip cache, hence bringing significant performance gain. Nonetheless, this inner-tile reuse[1] is not the whole story — in fact, $C[x]$ also has *inter-tile locality* across loop j while $B[y]$ has it across loop i. When the tile size is small or the cache size is large, it is also possible to exploit such inter-tile locality through the on-chip cache. This is especially the case when the tiled loop-nest is executed on a multithreaded-multicore processor; if each tile is executed by a single CPU thread, the inter-tile locality can be exploited when threads with such lo-

ASPLOS '17, April 8–12, 2017, Xi'an, China
© ACM. ISBN 978-1-4503-4465-4/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037709

[1] In this paper, we use locality and reuse interchangeably.

cality are grouped together and mapped onto the same core (so that they can share the same on-chip cache). This method to cluster threads for improved locality is known as *thread clustering* [4, 5].

On throughput-oriented processors such as GPU, the situation is similar but different. As shown in Listing 2, each GPU thread is responsible for one element in the $n \times n$ kernel grid space. Tiles still exist, but rather than mapping the entire tile to a single thread as in CPU, a tile is mapped to a *Cooperative Thread Array* (CTA) with each element in a tile assigned to a GPU thread of that CTA [6, 7]. An interesting question then arises: *is it possible to exploit the inter-tile locality, or the inter-CTA locality on GPU by applying a method similar to thread clustering used on CPU?*

A significant amount of prior efforts have been focused on developing performance enhancement techniques to exploit intra-CTA locality (e.g., intra-warp and inter-warp schemes) through warp-level scheduling [8–11] and cache contention/resource management [12–20]. While these works provide many useful insights on improving warp-level reference locality, none of them tackle the domain of inter-CTA locality exploitation on real GPU systems for potential performance benefits. Meanwhile, the latest GPU architecture design has already shown the trend where CTAs per SM increases but the number of hardware warp slots per SM remains the same. This further suggests the importance of exploiting such locality opportunity on GPU. However, there are three major obstacles hindering the research advancement in this domain:

- **Unclear Hardware Feasibility**. Existing GPU documentation and programming guides suffer from significant omissions and ambiguities on the subject of inter-CTA locality. No existing works have demonstrated if inter-CTA locality is exploitable on L1 or L1/Tex unified cache on real hardware, forcing performance optimization to be guided by folklore assumptions or research simulators [21]. For instance, common belief suggests that the lifetime and accessibility of data in L1 are restricted within the CTA boundary, similar to shared memory usage; while inter-CTA locality, if exists, is to be exploited at L2 level [11, 22].
- **Unknown CTA-Scheduler.** The default CTA scheduler, known as *GigaThread Engine* [23], is hardware implemented [24]. It is unknown and inaccessible [25]. Since no software approaches can tune or influence it directly [26], inter-CTA locality has been largely ignored by application developers due to lack of knowledge on how CTAs are assigned to SMs. It is also why the few existing CTA scheduling techniques are mostly hardware-based [8, 27].
- **Small Cache Capacity.** From performance optimization perspective, the CTA size (e.g., blockDim in CUDA) is often hard to change due to user and algorithm specification. As a result, the cache capacity should be sufficiently large to cover the data set of an entire CTA. However, GPUs have extremely small L1 and L1/Tex unified caches. For instance, while a Kepler's SM can accommodate 2k threads, by default it only has 16KB L1, leaving each thread with just 8 bytes cache space, too small for inter-CTA reuse.

```
//Loop1:Matrix-Vector-Multiply
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      C[i] += A[i][j] * B[j];

//Loop2:Tiling both i and j by m
  for (i=0; i<n; i+=m)
    for (j=0; j<n; j+=m)
      //The m*m Tiles
        for (x=i; x<i+m; x++)
          for (y=j; y<j+m; y++)
            C[x] += A[x][y]*B[y];
```

Listing 1: CPU Loop Tiling

```
__global__ void MMM(A,B,C){
//CTA coordinate in grid
  bx = blockIdx.x; by = blockIdx.y;
//Thread coordinate in CTA
  tx = threadIdx.x; ty = threadIdx.y;
//Thread coordinate in grid
  idx = bx * blockDim.x + tx;
  idy = by * blockDim.y + ty;
  P=A[idy][idx]*B[idx];
//atomic update or parallel reduction
  atomicAdd(&C[idy],P);}
 MMM<<<(n/m,n/m), (m,m)>>>(A,B,C);
```

Listing 2: GPU Kernel and CTA

To overcome these challenges and tap into the performance potential of inter-CTA reuse, we propose the concept of *CTA-Clustering* which aims to cluster CTAs with potential inter-CTA locality together and execute them concurrently or consecutively on the same SM. We develop its theory and explore design choices for its three internal steps — Partitioning, Inverting and Binding. Based on our empirical observation that inter-CTA locality can be harvested at L1 or L1/Tex unified cache, we propose an agent-based clustering scheme along with its complementary optimizations to enable efficient CTA-Clustering on real GPU hardware through *circumventing* the default hardware CTA scheduler. *To the best of our knowledge, this is the first work that provides effective software-based schemes to exploit inter-CTA locality on real GPU systems for immediate performance enhancement*. Specifically, this paper makes the following contributions:

- We empirically demonstrate that modern GPUs have the capability to exploit spatial and temporal inter-CTA locality on L1 or L1/Tex unified cache. (Section 3.1).
- We discover that inter-CTA reuse accounts for a significant portion of the global data reuse. We discuss if such reuse is exploitable based on its source of origin (Section 3.2).
- We propose the concept, methodology and design of CTA-Clustering (Section 4.2-4.3).
- We present an orchestrated CTA-Clustering framework to automatically exploit inter-CTA locality for general applications. The framework can be integrated as part of the compiler and immediately deployed on commodity GPUs (Section 4.4).
- We validate our designs on all modern generations of NVIDIA GPU architectures. Experiment results demonstrate that our proposed techniques can lead to significant performance speedup by substantially improving L1 hit rate and reducing L2 transactions (Section 5).

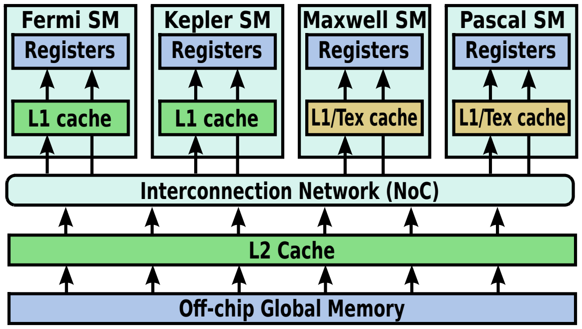


Figure 1: Architecture diagram for modern NVIDIA GPU architectures: Fermi, Kepler, Maxwell and Pascal. The arrows represent different global memory read datapaths for L1 and L2 data caches.

2. Background

Overall Architecture: A GPU processor consists of several SIMD cores, known as *Streaming Multiprocessors* (SMs) in CUDA terminology and *Compute Units* (CUs) in OpenCL[2]. Figure 1 shows the general architecture diagram for modern NVIDIA GPUs. In general, an SM contains scalar processors, register files, special function units, load-store units, shared memory and different types of caches. Table 1 shows the basic architectural specifications of four product examples. Each SM on Fermi and Kepler GPUs has a 48KB configurable, 128B cache-line, write-evict L1 data cache for general off-chip DRAM access [7]. It shares the same chip storage with the shared memory. However, recent Maxwell and Pascal GPUs devote such storage completely for shared memory, while relying on the texture cache (32B cache-line, non-coherent) to offer L1 caching capability, known as L1/Tex unified cache [28]. Meanwhile, all SMs in a GPU are connected via a NoC to a shared L2 cache. The L2 cache is banked and is writable (write-back and write-allocate [29]), having a cache-line size of 32B. Hardware cache prefetching is traditionally not enabled for GPUs [30]. However, one can choose to enable/disable the L1 cache via specific compiler options [7]. Note that *the L1 cache line size is larger than or equal to that of L2*. This is important for later discussion.

Execution Model: GPU SMs follow the *Single-Instruction-Multiple-Threads* (SIMT) execution model [6]. A group of 32 threads forms an execution vector, called *warp*. Warp is the basic unit for SM instruction fetching and decoding, with all threads inside a warp proceeding in a lock-step manner. Warps are registered in the warp-slots of an SM (Table 1). Several warps constitute a block called *thread block*, or *Cooperative-Thread-Array* (CTA), which encapsulates all the thread synchronization and barrier operations. CTA is the basic unit for delivering jobs to SMs. From the hardware perspective, there should be no dependency among CTAs — a kernel should always obtain correct result in arbitrary dispatching/execution order of CTAs. This feature ensures that applications remain unchanged when the CTA scheduling policy and/or the SM architecture is modified. Such an "*orderless*" property is also critical for this paper: *CTA scheduling can be manipulated without jeopardizing consistency*.

```
1  __global__ void kernel(float *input,int *smids,int *ticks){
2     int bid=blockIdx.x; int tid=threadIdx.x;
3     __shared__ float s_tmp=0; float tmp=0; unsigned t1,t2,smid;
4     asm("mov.u32 %0, %%smid;":"=r"(sm_id));//get SM id
5     unsigned idx=32*sm_id;//avoid reuse across SMs
6     if(tid==0){           //only use the primary thread
7 #ifdef STAGGERED
8        t0=clock(); while(clock()-t0<DELAY*bid);//stagger
9 #endif
10       t1=clock();             //start timer
11       __syncthreads();        //avoid multi-issuing for Line 10 & 12
12       tmp=input[idx]+RANDOM_NUMBER;   //global memory access through cache
13       __syncthreads();        //avoid multi-issuing for Line 12 & 14
14       t2=clock();             //stop timer
15       s_tmp=tmp;              //avoid nvcc opt-off
16       smids[bid]=smid; ticks[bid]=t2-t1; //save access cycles
17  }}
18 //Fermi:  CTAS=SM(15)*CTA_slots( 8)*Turnarounds(4)=480
19 //Kepler: CTAS=SM(15)*CTA_slots(16)*Turnarounds(4)=960
20 //Maxwell:CTAS=SM(16)*CTA_slots(32)*Turnarounds(2)=1024
21 //Pascal: CTAS=SM(20)*CTA_slots(32)*Turnarounds(2)=1280
22 kernel<<<CTAS,32>>>(input,smids,ticks);
```

Listing 3: Microbenchmark for verifying spatial and temporal inter-CTA locality.

CTA Scheduling: The default CTA scheduling policy on GPU has been assumed as *round-robin* (RR) [11, 27, 31–33]: First, the CTA-scheduler (i.e., GigaThread Engine) assigns each SM with at least one CTA. If an SM still has sufficient resources (e.g., registers, shared memory, warp-slots, etc) to sustain extra CTAs, a second round of assignment will be conducted. This rounding-assignment process repeats until all SMs are saturated, either bounded by resources or hardware limitation [32]. After that, a new CTA will be assigned to an SM whenever an existing CTA retires. Note that the CTA scheduling is purely managed by the hardware-implemented GigaThread Engine and there is no explicit software strategy that can impact the default CTA scheduling, nor can modify how CTAs are dispatched to SMs. Additionally, once a CTA is assigned to an SM, it cannot be preempted or reassigned to another SM [7]. Such lack of control over the CTA scheduler causes major obstacles for leveraging scheduling to boost GPU performance [25, 26]. Finally, the RR scheduling is ultimately an assumption, since the actual scheduling algorithm in the GigaThread Engine has never been disclosed [25].

3. Understanding Inter-CTA Reuse on GPU

3.1 Feasibility to Exploit Inter-CTA Reuse On L1

We designed a microbenchmark (shown in Listing 3) to verify if the GPU L1 cache is capable of exploiting data reuse among CTAs that are dispatched onto the same SM, either simultaneously (*spatial locality*) or subsequently (*temporal locality*). The configurations of the evaluated GPU platforms are listed in Table 1, including *Fermi*, *Kepler*, *Maxwell* and *Pascal*. As described in Line 18-21, we initiate 480, 960, 1024, 1280 CTAs for the four platforms respectively, corresponding to 4, 4, 2, 2 turnarounds per SM. Each CTA contains one warp and only the primary thread is essentially utilized to avoid intra-warp coalescing and inter-warp conflicting effects within the CTA boundary. Since there is only one warp per CTA, all the hardware CTA slots of an SM

[2] We use NVIDIA terminology in this paper as our validation platforms are NVIDIA products. However, the proposed techniques are general and applicable to other types of GPUs.

Table 1: Experiment Platforms. "CC." is the compute capability. "Dri/Rtm" is the CUDA Driver & Runtime version."Warp slots" and "CTA slots" are the maximum number of warps and CTAs per SM. "Regs" is the number of registers per SM. "SMem" is the size of the shared memory per SM. For Fermi and Kepler GPUs, the L1 cache and shared memory are configurable.

GPUs	Architecture	CC.	SMs	Dri/Rtm	Warp slots	CTA slots	L1(KB)	L1 cache-line	L2(KB)	L2 cache-line	Regs(KB)	SMem(KB)
GTX570	**Fermi**	2.0	15	6.5/6.5	48	8	16/48	128B	1536	32B	32	48/16
Tesla K40	**Kepler**	3.5	15	6.0/6.0	64	16	16/32/48	128B	1536	32B	64	48/32/16
GTX980	**Maxwell**	5.2	16	7.5/7.5	64	32	48	32B	2048	32B	64	96
GTX1080	**Pascal**	6.1	20	8.0/8.0	64	32	48	32B	2048	32B	64	64

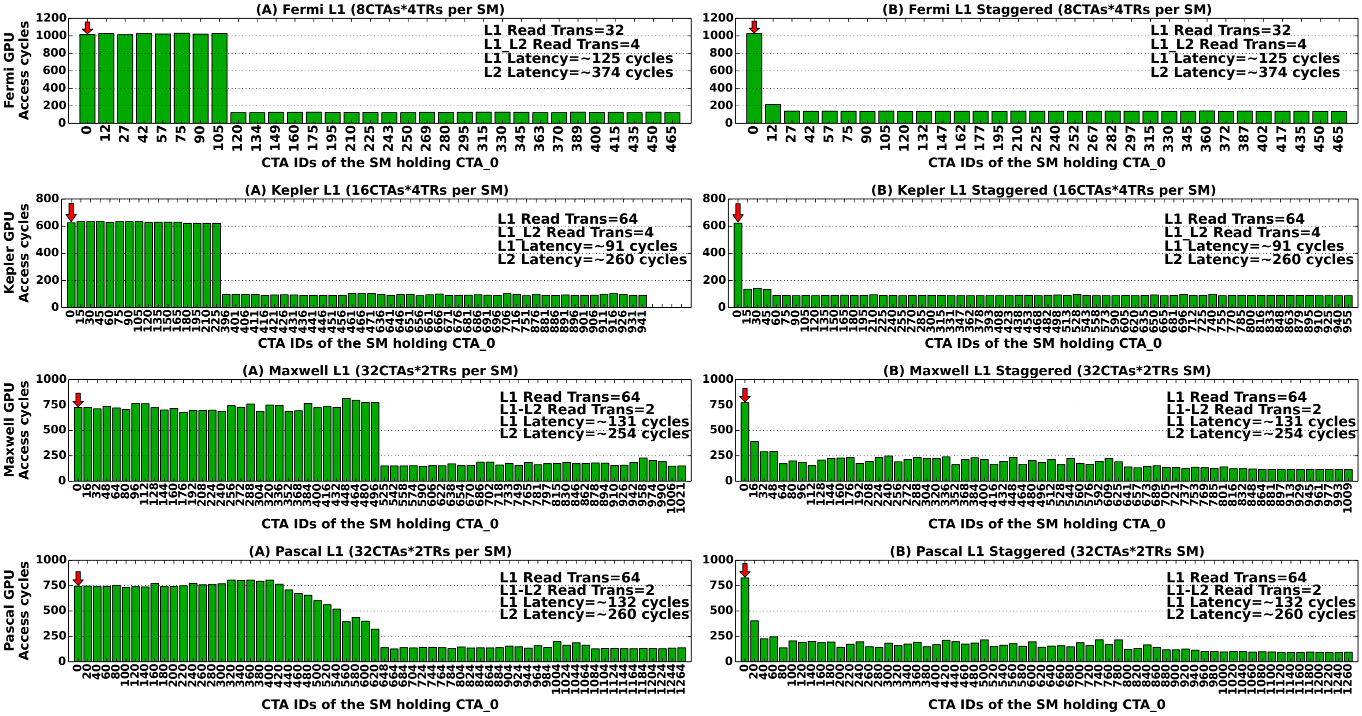


Figure 2: Exploiting inter-CTA reuse on the SM that holds CTA-0: (A) Default Scenario (temporal inter-CTA locality on L1); (B) Staggered Scenario (spatial inter-CTA locality on L1).

(Table 1) are essentially occupied. Thus, in each turnaround, an SM can simultaneously execute 8, 16, 32, 32 CTAs on the four platforms respectively. To measure the memory access latency more accurately, we also add two synchronization instructions before and after Line 12 when running on Maxwell and Pascal. This is because after the Maxwell generation, both the timing (Line 10 & 14) and the memory access (Line 12) instructions can be issued simultaneously, causing the timing results skewed. The CTA barriers (*__syncthreads()*) are used to disable multi-issuing.

Figure 2 shows the average memory access delay for the CTAs dispatched to a particular SM that accommodates the first CTA (i.e., CTA-0) under two setups: (A) default CTA scheduling, and (B) staggered execution. We denote this SM as "SM_0". The x-axis represents the CTA-IDs dispatched to SM_0. We also use CUDA profiler to profile two extra cache metrics (i.e., L1_Read_Trans and L1_L2_Read_Trans). We summarize the observations as follows:

(1) Temporal Locality: Figure 2-(A)s demonstrate that temporal locality can be exploited by subsequent turnarounds of CTAs at the L1 level (i.e., after a CTA is retired). This is proved by the observation that only CTAs in the first turnaround experience high access latency caused by the global memory loads (e.g., ∼750 cycles on Pascal). All the subsequent CTAs benefit from the locality on L1 generated in the first turnaround, so their access delay is similar to the L1 access latency (e.g., ∼132 cycles on Pascal). Note that for the multiple CTAs in the first turnaround, only one or two of them are essentially fetching data from off-chip DRAM; other CTAs, despite hit in L1 cache, are in fact *hit reserved* [21] (hit but the requested data is still on-the-fly) and thus exhibit similarly long delay. This is also confirmed by L1_Read_Trans and L1_L2_Read_Trans reported by the profiler: for Fermi and Kepler, one 128B L1 miss is equivalent to four 32B L2 read transactions; while for Maxwell and Pascal, the L1/Tex unified cache is partitioned into two *sectors* with each sector generating a 32B L1 miss, leading to two L2 read transactions. It is speculated that these sectors are private to particular CTA-slots following certain mapping mechanism.

(2) Spatial Locality: To verify whether spatial locality can be exploited at L1 cache by the CTAs that are dispatched to the same SM simultaneously (i.e., in the same turnaround), we enable *staggered execution* (Line 7-9 in Listing 3) to disalign their memory accesses so that the simultaneous memory requests from the concurrent CTAs cannot be aggre-

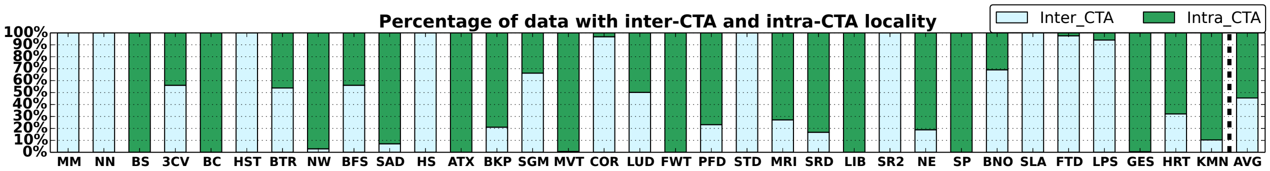
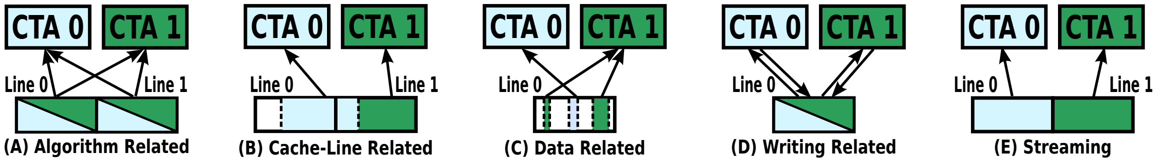


Figure 3: The percentage of intra- and inter-CTA reuse for common GPU applications.



(A) Algorithm Related (B) Cache-Line Related (C) Data Related (D) Writing Related (E) Streaming

Figure 4: The five application categories based on their sources of inter-CTA locality.

gated in L1 as previously. The *DELAY* variable in Line 8, which controls the degree of staggering, is set to be long enough (e.g., 1200 cycles) so that the data requested by the previous CTA can arrive in L1 before other CTAs in the same turnaround fetch. This also keeps all the concurrent CTAs alive and active. Figure 2-(B)s demonstrate such spatial reuse among the simultaneously launched CTAs. For example, only the first CTA in the first turnaround exhibits a much longer delay than the rest.

(3) Observed Hardware CTA Scheduling Policy: From the experiments on the four GPUs in Table 1 plus GTX750Ti (a first generation Maxwell GPU with CC-5.0), we found that *the default underlying CTA scheduler varies across different architectures*. We observed two general scheduling patterns based on the microbenchmark runs: (1) For the architectures in Table 1, CTAs in the first turnaround generally follows round-robin (RR), but not for those in the remaining turnarounds. They follow a more "demand-driven" policy. (2) On GTX750Ti, CTAs are randomly assigned to SM_0 within each individual turnaround instead of following any specific rule. Further investigations on real-world GPU applications confirm that the default hardware CTA scheduler is actually close to pattern (2) where the first turnaround does not necessarily follow RR either. We also find that the workload distribution is not balanced across SMs, even if the number of SMs can exactly divide the CTA number. For instance, on Kepler GPU shown in Figure 2-(A), the SM_0 only executes 60 CTAs rather than the expected 64.

3.2 Inter-CTA Locality Quantification and Sources

We quantify the percentage of the inter-CTA reuse among the total global data reuse, which is based on the data reuse of all the memory requests generated from SMs before they enter L1 cache. Since the current GPU profilers cannot track the address and source (e.g., CTA-id) of a specific memory request, for demonstration purposes only, we use GPGPU-Sim [21] to track the data reuse of all memory access requests and estimate the percentage of inter-CTA reuse among the overall data-reuse. Note that this estimation is data-driven and is independent of cache design or CTA-scheduling policy. The results are consistent from run to run. Figure 3 quantifies the inter-CTA reuse for 33 common GPU applications. It clearly demonstrates that the inter-CTA reuse in these applications accounts for a very significant portion of the overall data-reuse (on average 45%). This indicates that strategies aiming to improve inter-CTA reuse may lead to dramatic overall performance benefit, which goes beyond the conventional approaches that only attempt to promote intra- and inter-warp data reuse [16, 34–36].

However, *are these significant portions of inter-CTA locality exploitable on GPU hardware for enhancing performance?* To answer this, we need to understand where the inter-CTA locality originates from. Based on the characterization of a wide spectrum of GPU applications (see Table 2 for details), we classify the sources of GPU inter-CTA locality into the following five categories. Their patterns are depicted in Figure 4.

(A) Algorithm Related: Their inter-CTA locality is sourced from specific algorithm designs, in which certain data is utilized more than once by threads from different CTAs (Figure 4-(A)). For these applications, hints from algorithm designers are important for performance optimization. Besides, these applications often present promising opportunities for inter-CTA reuse. Typical examples include `MM`, `KMN`, `DCT`.

(B) Cache-line Related: Their inter-CTA locality is introduced by GPU cache design, or more specifically, the long cache-line sizes [17, 27]. As shown in Table 1, for Fermi and Kepler, a single cache miss from an integer access (4B) of a thread has to fetch an entire cache line of 128B into L1. As a result, the other 31 integers in this cache-line can be potentially accessed by threads from a different CTA (Figure 4-(B)). This scenario occurs when the memory access behavior is not perfectly coalesced or fully aligned with the cache-line boundary (e.g., *threads accessing halo region or user-defined object array*) and is especially prevalent in architectures having large L1 cache-line sizes (e.g., Fermi and Kepler). Typical examples include `SYK`, `NBO`, `ATX`.

(C) Data Related: Applications in this category are mostly dealing with irregular data structures, e.g., graphs, trees, hashes, pointer lists, etc. Inter-CTA locality comes from data organization in storage, or how data is accessed in memory (Figure 4-(C)). Due to the irregularity nature of data organization, such locality is often achieved by accident. Typical data-related applications include BFS, HST, BTR.

(D) Write Related: Applications in this category may have inter-CTA locality. However, as the GPU L1 cache adopts a write-evict policy [7, 29], the data that can be potentially reused could be evicted earlier by irrelevant writing from another CTA to the same cache-line (Figure 4-(D)). It happens when a kernel reads and writes to the same array, but having the access distance less than the cache-line size (e.g., the cache-line containing $a[i+1]$ is evicted when another CTA writes to $a[i]$). NW is a typical write-related application.

(E) Streaming: Memory accesses of streaming applications are mostly coalesced and aligned (Figure 4-(E)), while data is utilized only once or reused only within a CTA scope (e.g., via shared memory). These applications rarely have inter-CTA reuse. Typical streaming applications include BS, SAD, DXT.

By leveraging these application signatures, we propose software-based strategies to effectively exploit applications' inter-CTA locality at L1 level on commercial GPUs. We describe the methodology of our design next.

4. Design Methodology

4.1 Overall Strategy Outline

As discussed previously, some applications have ample inter-CTA locality (e.g., algorithm related) while others do not (e.g., streaming). Based on whether an application has *exploitable* inter-CTA locality, we propose a software approach for manipulating CTA scheduling on GPU, namely *CTA-Clustering*. Here we define whether an application has "*exploitable* " inter-CTA locality as follows:

- Locality from *algorithm related* (program defined) and *cache-line related* (architecture defined) applications can be identified before runtime thus is exploitable.
- Locality from *data related* (data defined), *write related* (has locality but cannot be utilized) and streaming (no locality) is either insignificant or can only be determined at runtime. Thus it is either impossible or very little benefit to exploit. We consider them having no exploitable inter-CTA locality.

Note that for some data-related applications with very specific runtime access patterns, their data organization in memory may be predicted before runtime [37], making their inter-CTA locality exploitable. For example, previous works [38, 39] have suggested to use a lightweight *inspector kernel* before runtime to profile local data access of certain graph-processing applications (e.g., first few layers of BFS), in order to predict global data organization for optimizing runtime access.

Figure 5 outlines the overall flow for exploiting inter-CTA locality on GPU. For applications having exploitable inter-CTA locality, we apply CTA-Clustering to maximize their inter-CTA data reuse on L1 (Section 4.2 and 4.3-(I)(II)). For applications having no exploitable inter-CTA locality, we first apply CTA-Clustering to reshape the default CTA scheduling pattern (not for exploiting inter-CTA locality but to impose a certain CTA execution order), and then use CTA-Prefetching to preload data that is required by the upcoming CTAs before the current one expires (Section 4.3-(III)).

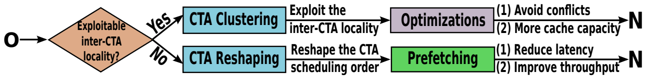


Figure 5: Optimization Strategy. O stands for original kernel. N stands for new kernel.

4.2 CTA Clustering

The basic idea of CTA-Clustering is to initiate a new kernel to replace the original one so that a predefined clustering rule can be established. The objective is to cluster CTAs with inter-CTA locality together and execute them concurrently or consecutively on the same SM. We use $\boldsymbol{O}, \boldsymbol{N}, \boldsymbol{C}$ to denote the **O**riginal kernel, the **N**ew kernel and the **C**lusters. CTA-Clustering is essentially to find a mapping: $\boldsymbol{N} \rightarrow \boldsymbol{O}$ subjects to $\boldsymbol{C}$, shown in Figure 6. The process comprises three steps:

Step-1. Partitioning: Partition the CTAs of $\boldsymbol{O}$ into M balanced clusters $\boldsymbol{C}$, with the most inter-CTA locality conserved within each cluster. It is defined by $\boldsymbol{f}$ in Figure 6.

Step-2. Inverting: Reconstruct $\boldsymbol{O}$ from $\boldsymbol{C}$. In other words, given a CTA in a certain cluster, we can retrieve its CTA id in the original kernel $\boldsymbol{O}$. Inverting is defined by $\boldsymbol{f^{-1}}$ in Figure 6.

Step-3. Binding: Bind the CTAs of the new kernel $\boldsymbol{N}$ to the clusters $\boldsymbol{C}$. Assume the size of $\boldsymbol{C}$ is equal to the number of SMs, the mapping from $\boldsymbol{C}$ to SMs is a 1-to-1 mapping. Binding is defined by $\boldsymbol{g}$ in Figure 6.

In short, finding the mapping $\boldsymbol{N} \rightarrow \boldsymbol{O}$ follows:

$$\boldsymbol{O} \underset{\text{Step-2: Inverting}}{\overset{\text{Step-1: Partitioning}}{\rightleftharpoons}} \boldsymbol{C} \xleftarrow{\text{Step-3: Binding}} \boldsymbol{N} \qquad (1)$$

To clearly demonstrate the theory, we showcase the clustering process using a well-known application —*Matrix Multiplication* (MM) from CUDA SDK [40]. Based on the source code of MM, its *intra-CTA data reuse* is completely handled by shared memory (i.e., local buffers for matrix A and B are declared in the shared memory for intra-CTA data sharing between threads)— inter-CTA locality is not explicitly explored. However, as depicted in Figure 8-(A), MM inherently has algorithm-related inter-CTA locality, which could be potentially exploited for performance enhancement. Due to space limitation, we do not show MM's source code here. We will refer it as *MM_Kernel_Body* throughout this section.

4.2.1 Step 1. CTA-Partitioning: $\boldsymbol{f} = \boldsymbol{O} \rightarrow \boldsymbol{C}$.

The formal definition of the *partitioning problem* is:

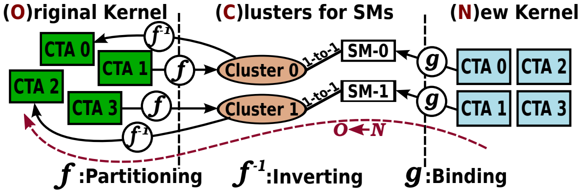


Figure 6: CTA Clustering is to find $N \rightarrow O$ (red dot-line) or $f^{-1}(g(\text{new kernel CTA id}))$.

Problem 1 *Given an undirected graph $G(V, E_\mu)$ with each uniform vertex represents a CTA and each weighted edge represents the inter-CTA reuse with degree μ, partition the graph into M balanced clusters so that the weight summation of the inner-cluster edges are maximized.*

We define $y_{v,i} = 1$ if vertex v is assigned to cluster i ($i \in [M]$); otherwise $y_{v,i} = 0$. Consequently, we can define cluster i as: $\boldsymbol{C_i} = [(v,e)|v \in V, y_{v,i} = 1]$. The partitioning problem then can be formulated as:

$$max\ (\sum_{i=1}^{M} \sum_{e_u \in C_i} \mu) \quad subjects\ to$$

$$\forall v \in V \mid \sum_{i=1}^{M} y_{v,i} = 1 \ \textit{//one vertex assigned to one cluster}$$

$$\forall i \in [M] \mid \sum_{v \in V} y_{v,i} = |V|/M \ \textit{//clusters are balanced}$$

The general balanced-graph partitioning is NP-complete [41]. However, it is possible for the application developers to generate a desired partition function based on their knowledge of the algorithm. In this work, we provide a heuristic solution that ensures clustering balance as part of an automatic framework (Section 4.4). We define the partition function as:

$$\boldsymbol{f(v) = v \rightarrow (w,i)} \mid v \in V,\ i \in [M],\ w \in [|\boldsymbol{C}_i|] \qquad (2)$$

where $\boldsymbol{v}$ is the CTA id in $\boldsymbol{O}$, $\boldsymbol{w}$ is the id in $\boldsymbol{C_i}$. For instance, if a cluster contains 3 CTAs, $w \in [0,1,2]$. As V defines the order of CTA in $\boldsymbol{O}$ and $|V|$ is the quantity of CTAs in $\boldsymbol{O}$, our solution is to separate $|V|$ into M chunks via:

$$f(v) = (w,i) = (v\%(|V|/M),\ v/(|V|/M)) \qquad (3)$$

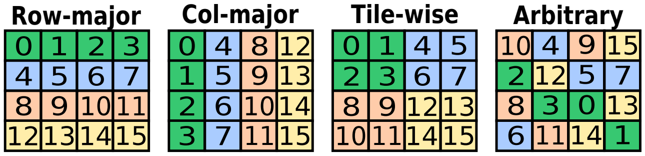
However, $|V|$ is not necessarily the multiples of M. In order to distribute the chunks as balanced as possible, we extend Eq. 3 into a conditional equation: if $v\%M \leq |V|\%M$,

$$f(v) = (w,i) = (v\%(|V|/M+1),\ v/(|V|/M+1)) \qquad (4)$$

Otherwise,

$$f(v) = (w,i) = (v\%(|V|/M), (v-|V|\%M)/(|V|/M)) \qquad (5)$$

Different from the conventional wisdom [27, 36], our partition approach does not simply separate CTAs into consecutive chunks because vertex v's order is completely dictated by how CTAs are indexed in $\boldsymbol{O}$. Figure 7 shows four major CTA indexing methods for a 2D grid. For the default *row-majored* indexing, if using CUDA terminology, $v =$ `blockIdx.y` * `gridDim.x` + `blockIdx.x`, it clusters consecutive or row-adjacent CTAs. In other words, it partitions CTAs along the Y-dimension (i.e., Y-partitioning). If the *column-major* indexing is adopted (i.e., $v =$ `blockIdx.x` * `gridDim.y` + `blockIdx.y`), CTAs will be partitioned along the X-dimension (i.e., X-partitioning). Moreover, the *Tile-based* 2D indexing can partition CTAs along both X- and Y-dimensions. However, tile-based indexing may also incur higher overhead due to complex index calculation (Section 5.2). Finally, it is also possible to cluster arbitrary CTAs by choosing a customized indexing method. Now recall Section 4.1, we describe the partition methods for the two types of applications that have exploitable inter-CTA locality as follows:



Row-major

🟩 0	🟩 1	🟩 2	🟩 3
🟦 4	🟦 5	🟦 6	🟦 7
🟧 8	🟧 9	🟧 10	🟧 11
🟨 12	🟨 13	🟨 14	🟨 15

Col-major

🟩 0	🟦 4	🟧 8	🟨 12
🟩 1	🟦 5	🟧 9	🟨 13
🟩 2	🟦 6	🟧 10	🟨 14
🟩 3	🟦 7	🟧 11	🟨 15

Tile-wise

🟩 0	🟩 1	🟦 4	🟦 5
🟩 2	🟩 3	🟦 6	🟦 7
🟧 8	🟧 9	🟨 12	🟨 13
🟧 10	🟧 11	🟨 14	🟨 15

Arbitrary

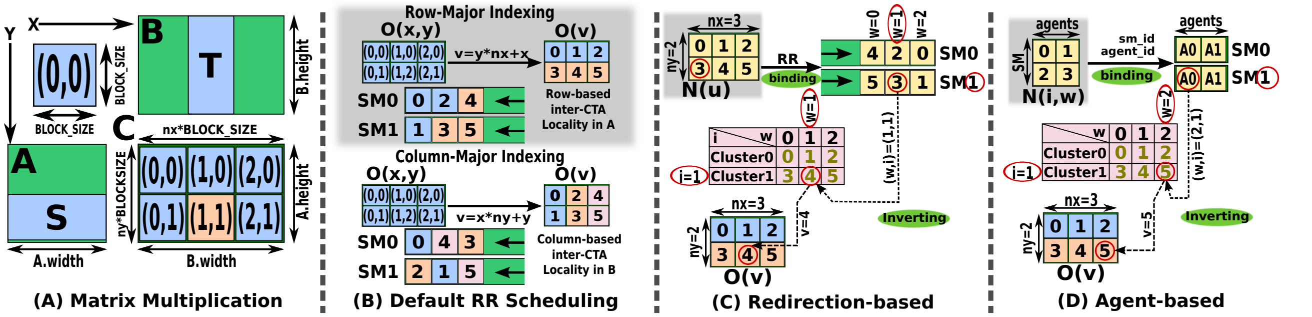
🟧 10	🟦 4	🟧 9	🟨 15
🟩 2	🟨 12	🟦 5	🟦 7
🟧 8	🟩 3	🟩 0	🟨 13
🟦 6	🟧 11	🟨 14	🟩 1

Figure 7: Four major CTA indexing methods for a 2D grid.

(A) Algorithm-Related: To make the partition process general and automatic, we propose a solution based on dependency analysis for the array references over the coordinates of different grid-dimensions, which is similar to the *dependency analysis* for a loop-nest [2]. If a kernel grid (*gridDim*) is 1D, we simply perform X-partitioning. If a kernel grid is 2D and the CTA X-direction based variable (e.g., *blockIdx.x*) is the last or the only dimension of an array reference (e.g., $A[\alpha(by) + bx + \varepsilon(tx,ty)]$ or $A[\beta(bx)]$), which indicates it may have inter-CTA locality across X, we then perform Y-partition using row-majored indexing. Otherwise, if the Y-based variable (e.g., *blockIdx.y*) remains the last or the only dimension of an array reference (e.g., $A[\alpha(bx) + by + \varepsilon(tx,ty)]$ or $A[\beta(by)]$), we perform X-partition using column-majored indexing. The partition process for 3D kernels is similar to 2D but with more options. However, most of the common GPU applications only contain 2D kernel grids.

(B) Cache-line Related: Since CUDA and C/C++ commonly adopt row-major policy to organize and store multi-dimensional arrays, the cache-line related inter-CTA locality often exists between row-adjacent CTAs. Thus we use row--major CTA indexing (Y-partition) to count the CTA order from $\boldsymbol{O}$.

As discussed in Section 4.1, some *data-related* applications can also exploit inter-CTA locality if a detailed customized partition is provided by users. But this is beyond of the scope of this work. Using `MM` as an example shown in Figure 8-(A), A and B have inter-CTA reuse in S and T region, respectively. But whether to partition the CTAs along Y (for locality in A) or along X (for locality in B) depends on if $A.height$ is larger than $B.width$ (i.e., directional locality intensity). In this case, we target on A using Y-partitioning (i.e., row-major indexing). Through the known parameters: $M = 2$, $grid\_width = 3$, $grid\_height = 2$ in Figure 8-(A), we have $|V| = 6$ and $|V|\%M = 0$, which is always $\leq v\%M$. Therefore, we can use Eq. 5 to obtain the partition function



(A) Matrix Multiplication

(B) Default RR Scheduling

(C) Redirection-based

(D) Agent-based

Figure 8: CTA-Clustering Process using MM as an example. (A): CTA(1,1) has inter-CTA reuse with CTA(0,1) and CTA(2,1) in matrix-A since they all load region S from A. Meanwhile, CTA(1,1) has inter-CTA reuse with CTA(1,0) in matrix-B since they both load region T from B. (B): The default RR scheduling policy assumed by previous works cannot preserve inter-CTA locality, regardless if it is in default row-major indexing or column-major indexing. (C): Redirection-based Clustering binds CTA u of new kernel to (w,i) using RR assumption. It then uses (w,i) to locate CTA v of the old kernel through inverting. (D): Agent-based Clustering enables persistent CTAs (active agents) per SM to process the (w,i) task list. It binds u of new kernel to (w,i) via register fetching for sm_id and agent_id. It then uses the two ids to calculate (w,i) and uses (w,i) to locate v of the old kernel through inverting.

$\boldsymbol{f(v)}$ and use it to locate which cluster an arbitrary CTA v in $\boldsymbol{O}$ should be dispatched to. For instance, CTA-(0,1) can be located as the 0th element in cluster C_1 via:

$$f(\text{CTA-(0,1)}) = f(\text{CTA-id=1*3+0}) = f(v=3)$$
$$= (3\%(6/2), (3-6\%2)/(6/2)) = (0,1) = (w,i)$$

4.2.2 Step 2. CTA-Inverting: $\boldsymbol{f^{-1} = C \rightarrow O}$.

Since Eq. 3 is a *one-to-one* mapping function, we can obtain its **inverse function** as

$$\boldsymbol{v = f^{-1}(\ (w,i)\ )} = i*(|V|/M)+w \qquad (6)$$

Similarly, the inverse functions of Eq. 4 and 5 are

$$\begin{cases} \forall\, i \leq |V|\%M:\ v = i*(|V|/M+1)+w \\ \forall\, i > |V|\%M:\ v = i*(|V|/M+1)+w+|V|\%M-i \end{cases}$$

The two equations can be unified as

$$v = i*(|V|/M+1)+w+min(|V|\%M-i,0) \qquad (7)$$

Given a pair of (w,i) (a cluster i and a position in this cluster w), its corresponding CTA id v can be located in $\boldsymbol{O}$ through Eq. 7. If the indexing method is also known, we can further obtain its coordinate in $\boldsymbol{O}$. For MM, a CTA labeled $(w=2, i=1)$ in $\boldsymbol{C_1}$ can find its corresponding v in $\boldsymbol{O}$ via:

$$v = f^{-1}(\ (2,1)\ ) = 1*(6/2+1)+2+min(0-1,0) = 5$$

4.2.3 Step 3. CTA-Binding: $\boldsymbol{g = N \rightarrow C}$.

The formal definition of the *binding problem* is:

Problem 2 *Given the new kernel* $\boldsymbol{N}$*, how to associate the CTAs in* $\boldsymbol{N}$ *to the clusters* $\boldsymbol{C}$ *so that all the items in* $\boldsymbol{C}$ *can be completely and precisely executed?*

The objective of this step is to find the mapping from $\boldsymbol{N} \rightarrow \boldsymbol{C}$, or essentially $\boldsymbol{N} \rightarrow (w,i)$. In other words, given an arbitrary CTA u in the new kernel $\boldsymbol{N}$, how to learn the job target (w,i) it is responsible for. Since the GigaThread Engine is unknown and inaccessible, depending on how (w,i) is obtained in the context of $\boldsymbol{N}$, we propose two methods to *trick or circumvent* the hardware CTA scheduler:

(A) RR-based Binding: This simple approach is to obtain (w,i) in $\boldsymbol{N}$ based on the assumption that the GigaThread Engine follows a strict-RR policy in the new kernel. With this assumption, we can calculate the mapping of $\boldsymbol{N} \rightarrow \boldsymbol{C}$ via:

$$(w,i) = (u/M, u\%M) \ \mid\ u \in \boldsymbol{N},\ (w,i) \in \boldsymbol{C} \qquad (8)$$

In MM, the associated (w,i) for a CTA in $\boldsymbol{N}$ with its id$=4$ can be calculated via Eq. 8: (w,i)=(4/2, 4%2)=(2,0)

(B) SM-based Binding: Unlike RR-based binding, this approach has no preassumption on the default CTA scheduling policy. Assuming *Cluster_i* is binded to *SM_i*, we need to learn how the CTAs allocated by the default scheduler on *SM_i* are mapped to *Cluster_i*, by obtaining their corresponding (w,i). To identify the corresponding *Cluster_i* at runtime, a CTA can fetch the physical id of the SM it currently locates on from a special register [25]:

```
asm("mov.u32 %0, %%smid;":"=r"(sm_id));//obtain i
```

To identify its position w in a cluster, a CTA has to synchronize with the other CTAs on the same SM to avoid conflicts (i.e., two CTAs obtaining the same w). We discovered that on Fermi and Kepler the way CTAs are binded to *hardware warp-slots* is essentially consecutive and fixed. Thus a CTA can decide its position w in the cluster relying on its hardware warp-slot id divided by *WARPS_PER_CTA*, as shown in Line 5-6 in Listing 5. However, on Maxwell and Pascal, warps from different CTAs are dynamically binded to hardware warp-slots [42]. Thus we rely on a global atomic operation and shared memory broadcasting (Line 16-19 in Listing 5).

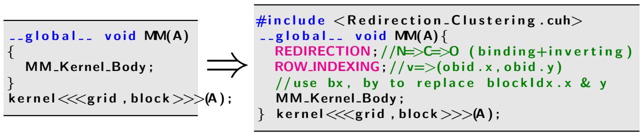
4.2.4 Putting It All Together.

Combining all the three steps, we propose two CTA-Clustering approaches based on the two binding schemes:

(1) Redirection-based Clustering: Shown in Figure 8-(C), this design is built on RR-based binding. The number of CTAs in the new kernel is identical to the old kernel (i.e., $|\boldsymbol{N}| = |\boldsymbol{O}|$, 1-to-1 mapping from $\boldsymbol{N}$ to $\boldsymbol{O}$). "Redirection" means that each CTA u in $\boldsymbol{N}$ is redirected to a CTA v in $\boldsymbol{O}$. We implement the idea (Eq. 7 and 8) into a header file

```
//======================== Redirection_Clustering.cuh =========================
#define REDIRECTION const int _ctas=gridDim.x*gridDim.y;\ //CTA num of O and N
const int u=blockIdx.y*gridDim.x+blockIdx.x;\ //row-major indexing in N
const int v=(u%SM)*(_ctas/SM+1)+u/SM+min(0,(_ctas%SM)-(u%SM));//binding+inverting
#define COL_INDEXING int bx = v/gridDim.y; int by = v%gridDim.y;
#define ROW_INDEXING int by = v/gridDim.x; int bx = v%gridDim.x;
```

Listing 4: Redirection-based Clustering Header File



```
__global__ void MM(A)
{
  MM_Kernel_Body;
}
kernel<<<grid,block>>>(A);
```

```
#include <Redirection_Clustering.cuh>
__global__ void MM(A){
  REDIRECTION;//N=>C=>O (binding+inverting)
  ROW_INDEXING;//v=>(obid.x,obid.y)
  //use bx, by to replace blockIdx.x & y
  MM_Kernel_Body;
} kernel<<<grid,block>>>(A);
```

Figure 9: Kernel Transformation by Redirection-based Clustering

(Listing 4), based on which Figure 9 shows the simple code transformation using MM. Although this scheme is easy to implement and low-cost, it is built upon the assumption of strict-RR for the hardware CTA scheduler, which has been proven incorrect on real GPU hardware (Section 3.1-(3)). Several previous works have also assumed RR for CTA scheduling [11, 27, 31–33].

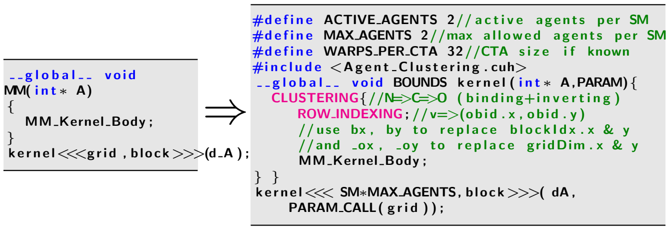
(2) Agent-based Clustering: This design (Figure 8-(D)) is built on SM-based binding scheme, and the volume of CTAs in $\boldsymbol{N}$ is *not* identical to $\boldsymbol{O}$. Different from tricking the GigaThread Engine in Redirection-based Clustering, Agent-based Clustering completely circumvents the hardware CTA scheduler. As illustrated in Figure 8-(D), we allocate a few CTAs that persistently reside on each SM during the kernel execution (similar to the concept of persistent threads [26, 43]), named "*agents*", serving all the (w,i) (or *tasks*) in the cluster belong to that SM via a task loop. For instance, agents A0 and A1 on SM-0 will executes all the CTAs in $\boldsymbol{O}$ that were clustered to $\boldsymbol{C_0}$ in the partitioning step. Note that the number of agents on a SM is normally way less than the size of the cluster, e.g., 16 agents vs. 2K CTAs. The inter-CTA locality is exploited when the agents of an SM are working in parallel (spatial locality), and among consecutive tasks (temporal locality). We also implement a header file for this design, shown in Listing 5, based on which a very simple code transformation using MM is shown in Figure 10. Although this design works regardless of the default CTA scheduler, it does associate with additional cost in the SM-based binding (i.e., SM id fetching, agent_id calculation and synchronization). We will discuss the performance impact of these overheads in Section 5.

4.3 Complementary Optimizations to CTA-Clustering

(I) CTA Throttling. CTA throttling limits the number of concurrent CTAs on an SM to reduce the contention for execution resources (e.g., caches and bandwidth). Previous works [27, 31] have observed that using maximum number of CTAs per SM is not always optimal. We enable software-based CTA throttling by controlling the number of the active agents per SM. It is achieved by directly retiring a CTA when its agent_id is larger than specified (line 7 in Listing 5). However, naively decreasing the total number of agents in kernel grid configuration to adjust throttling degree can cause incorrect execution due to the imbalanced agent dispatching among SMs from the hardware scheduler (Section 3.1-(3)). Thus we always allocate the maximum allowable agents per SM for an application (e.g., bounded by register usage and shared memory) at kernel grid configuration to occupy the CTA slots (*MAX_AGENTS* in Figure 10) so as to force balanced agent distribution, but only activate some of them at runtime (*ACTIVE_AGENTS* in Figure 10) depending on its agent id (Line 7 and 20 in Listing 5). In this way, we can control the throttling degree while guaranteeing agents are evenly distributed to SMs. Additionally, we leverage *'__launch_bounds__'* [7] to increase register usage during compilation when there are less CTAs than permitted. It exploits register reuse and hides instruction latency. Note that throttling is not a necessity for all applications. It is applied when the performance is not improved or even degraded after clustering (i.e., reducing capacity misses). To decide active_agents at runtime, we refer to a dynamic CTA voting scheme similar to that used in [12].

```
1 //========================== Agent_Clustering.cuh ============================
2 #if __CUDA_ARCH__ < 500 //Fermi and Kepler GPU
3 #define CLUSTERING \
4   int sm_id;asm("mov.u32 %0,%%smid;":"=r"(sm_id));\//fetch sm-id
5   int warp_id;asm("mov.u32 %0,%%warpid";:"=r"(warpid));\//fetch hardware warp id
6   const int agent_id=warp_id/WARPS_PER_CTA;\//only appicable to Fermi and Kepler
7   if (agent_id >= ACTIVE_AGENTS) return;\ //for CTA throttling
8   const int _jobs = _cd + ((sm_id<_ck)?1:0);\//workload per SM
9   const int _base = _cn*sm_id + (((_ck-sm_id)<0)?(_ck-sm_id):0);\//start position
10  for(int v=_base+agent_id;v<_base+_jobs;v+=ACTIVE_AGENTS)//traverse the cluster
11 #else //Maxwell and Pascal GPU
12 #define CLUSTERING \
13  int sm_id; asm("mov.u32 %0,%%smid;":"=r"(sm_id));\//fetch sm-id
14  const int _jobs = _cd + ((sm_id<_ck)?1:0);\ //workload per SM
15  const int _base = _cn*sm_id + (((_ck-sm_id)<0)?(_ck-sm_id):0);\//start position
16  __shared__ int agent_id;\ //on shared memory for broadcasting
17  if (threadIdx.x==0 && threadIdx.y==0)\ //only 1st thd to bid agent id
18    agent_id = atomicAdd(&global_counters[sm_id],1);\
19  __syncthreads();\ //other thds in the CTA wait for gaents_id
20  if (agent_id >= ACTIVE_AGENTS) return;\ //for CTA throttling
21  for(int v=_base+agent_id;v<_base+_jobs;v+= ACTIVE_AGENTS)//traverse the cluster
22 #endif
23 #define COL_INDEXING int bx=_v/_oy; int by=v%_oy; //partition along X dimension
24 #define ROW_INDEXING int by=_v/_ox; int bx=v%_ox; //partition along Y dimension
25 #ifdef WARPS_PER_CTA //for CTA throttling
26 #define BOUNDS __launch_bounds__ (32*WARPS_PER_CTA,WORKERS)//opt for reg usage
27 #else
28 #define BOUNDS //if CTA size is unknown, ignore this optimization
29 #endif
30 #define PARAM const int ctas, const int _cd, const int _ck, const int _cn, \
31  const int _ox, const int _oy
32 #define PARAM_CALL(X)  ((X).x*(X).y*(X).z), ((X).x*(X).y*(X).z)/SM, \
33  ((X).x*(X).y*(X).z)%SM, (((X).x*(X).y*(X).z)+SM-1)/SM+1, (X).x, (X).y
34 //Prefetching for Fermi & Kepler L1 cache
35 #define PREFETCH_L1(X) asm("prefetch.global.L1 [%0];"::"l"(&X));
36 //Prefetching for Maxwell & Pascal L1/Tex unified cache
37 #define PREFETCH_L1T(X)  __ldg((int*)&x);
```

Listing 5: Agent-based Clustering Header File



```
__global__ void
MM(int* A)
{
  MM_Kernel_Body;
}
kernel<<<grid,block>>>(d_A);
```

```
#define ACTIVE_AGENTS 2//active agents per SM
#define MAX_AGENTS 2//max allowed agents per SM
#define WARPS_PER_CTA 32//CTA size if known
#include <Agent_Clustering.cuh>
__global__ void BOUNDS kernel(int* A,PARAM){
  CLUSTERING{//N=>C=>O (binding+inverting)
    ROW_INDEXING;//v=>(obid.x,obid.y)
    //use bx, by to replace blockIdx.x & y
    //and _ox, _oy to replace gridDim.x & y
    MM_Kernel_Body;
} }
kernel<<< SM*MAX_AGENTS,block>>>( dA,
    PARAM_CALL(grid));
```

Figure 10: Kernel Transformation by Agent-based Clustering

(II) Cache Bypassing. A large body of work [12–16, 19, 20] has proposed cache bypassing techniques for GPU, aiming to avoid unnecessary cache pollution (e.g. severe thrashing), reduce capacity and conflict misses due to very limited cache capacity and resources (e.g., MSHRs and miss

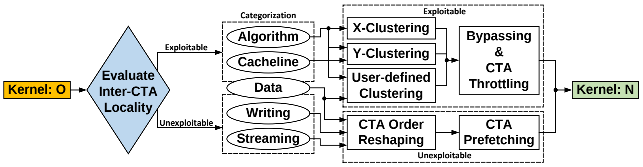


Figure 11: Inter-CTA Locality-aware Optimization Framework

queue slots). As a complementary technique to further enhance CTA-Clustering, we bypass the streaming accesses to L1 or L1/Tex unified cache to prevent them from contending resources with the accesses that have inter-CTA reuse.

```
//Bypassing for L1 and L1/Tex unified cache
asm("ld.global.cg.f32 %0, [%1];":"=f"(X):"l"(&Y));
```

(III) CTA Prefetching using Reshaped Order. As discussed in Section 4.1, CTA prefetching is for applications without exploitable inter-CTA locality (i.e., *data-related*, *writing-related* or *streaming*). For them, our core technique — CTA clustering is not expected to directly benefit performance from inter-CTA locality perspective. But CTA-Clustering (e.g., Y-partitioning) can impose a specific CTA scheduling order (i.e., reshaping the default order), which enables the current CTA to be able to preload the required data for its successor(s) in advance, thus hiding long memory access delay to L2 or DRAM. This is feasible because GPU L1 cache preserves data after a CTA is retired. Without this reshaped order, prefetching can only be limited within a CTA scope [32, 36, 44, 45] due to the "orderless" property of CTA dispatching. The macros for GPU software prefetching are listed in Line 34-37 in Listing 5.

4.4 Inter-CTA-Aware Optimization Framework

Figure 11 illustrates our software-based optimization framework for exploiting inter-CTA locality. The previous subsections have explained how to conduct the corresponding optimizations once we have certain knowledge about the target kernel (e.g., whether it has exploitable inter-CTA locality or not). In this section, we primarily discuss how the framework estimates an application's source of inter-CTA locality. This process is highlighted in blue in Figure 11.

The framework applies several simple coarse-grained techniques to estimate an application's source of inter-CTA locality before it is further analyzed for optimizations. Since intra-CTA locality is generally captured within the CTA scope (e.g., via shared memory), altering CTA scheduling will only impact inter-CTA locality. Thus we first apply the simple *Redirection-base Clustering* (Figure 9) to impose a new CTA execution order (either X- or Y-clustering) and check whether the performance and/or L1 hit rate have changed (an indicator for inter-CTA locality potential). The kernels with higher potential (e.g., significant change in L1 hit rate) may belong to *algorithm-related* or *cache-line related*. Note that it is better to reduce the problem size (total CTA number) for this verification because large CTA number per SM can trigger severe trashing, often leaving L1 hit rate close to zero. Furthermore, we can turn on/off L1 cache to see if there is any performance change. If L2 cache access is significantly reduced after turning off L1, it is likely caused by large L1 cache-line which hints *cache-line related*. If there is no change in L2 transactions but the coalescing degree reported by the CUDA profiler is high, this kernel is probably *streaming*. However, if the coalescing degree is low, the memory access behavior is likely to be more random, suggesting *data-related*. Finally, the framework evaluates the references to array inside the kernel. If the kernel reads and writes to the same array but with shifted or skewed references, it is likely that the written results can be reused later for other CTAs, suggesting *write related*.

5. Evaluation

5.1 Overall Result Summary

We evaluate the proposed CTA-Clustering techniques along with the three complementary optimizations on the four GPU architectures in Table 1. Shown in Table 2, we select twenty-three representative applications from popular GPU benchmark suits, covering the five application categories in terms of inter-CTA locality sources (Section 3.2). Figure 12 shows the overall performance results of these applications using our proposed schemes. The results are demonstrated based on architecture and application type. Each GPU architecture (each row) contains three sub-figures: (1) the **left** one represents *algorithm related* applications (marked in Table 2); the **middle** one represents *cache-line related* applications; and the **right** one consists of *data related*, *write related* and *streaming* applications that do not have exploitable inter-CTA locality. Additionally, we include the change of *Achieved Occupancy* [49] (*AC_OCP*) in the figure to better demonstrate the execution status. To understand the impact of our techniques on cache performance, we plot the *L2 cache transactions* and L1 hit rates in Figure 13. Compared to L1 hit rate, L2 cache transaction is a better performance indicator for GPUs [50].

Figure 12 shows that our proposed CTA-Clustering technique and its associated optimizations achieve an average performance speedup of 1.46x, 1.48x, 1.45x, 1.42x (up to 3.83x, 3.63x, 3.1x, 3.32x) for *algorithm-related* applications on Fermi, Kepler, Maxwell and Pascal GPUs, respectively. Meanwhile, they also achieve an average speedup of 1.47x and 1.29x (up to 2.57x and 1.75x) for *cache-line* related applications on Fermi and Kepler GPUs, which have a larger L1 cache-line size. Regarding the cache performance (Figure 13), our techniques reduce the L2 transactions by 55%, 65%, 29%, 28% for the *algorithm related* applications on the four architectures, respectively. For the *cache-line related* applications, we achieve 81%, 71% and 34% L2 cache transaction reduction on Fermi, Kepler and Maxwell, respectively.

5.2 Observations, Analysis and Limitations

Regarding the results in Figure 12 and 13, we have the following observations and analysis:

Table 2: Benchmark Characteristics. "WP" stands for warps per CTA. "CTAs" means the default number of CTAs per SM in baseline. The four items in "6/8/8/8" are the values for Fermi/Kepler/Maxwell/Pascal. Similar meaning applies to other columns in this table. "Registers" indicates the register cost per thread in baseline. "SMem" is the shared memory cost per CTA in baseline. "Partition" is the partition method adopted during CTA clustering. "Opt Agents" is the number of optimal agents for CTA throttling. "Ref" refers to the sources.

Application	Description	abbr.	Category	WP	CTAs	Registers	SMem	Partition	Opt Agents	Ref.
kmeans	Clustering algorithm	`KMN`	Algorithm	8	6/8/8/8	14/17/16/18	0	X-P	1/1/1/1	[46]
matrixMul	Matrix multiplication	`MM`	Algorithm	32	1/2/2/2	22/29/32/27	8192B	Y-P	1/2/2/2	[40]
nn	Convolutional neural network	`NN`	Algorithm	1	8/16/32/32	21/35/37/32	0	Y-P	8/16/32/32	[21]
imageDenoising	NLM method for image denoising	`IMD`	Algorithm	2	8/16/18/18	63/61/49/55	0	Y-P	8/16/14/16	[40]
backprop	Perception back propagation	`BKP`	Algorithm	8	6/8/8/8	11/11/16/18	1092B	X-P	6/8/8/8	[46]
dct8x8	Discrete cosine transform	`DCT`	Algorithm	2	8/16/32/32	14/17/22/19	512B	X-P	8/16/32/24	[40]
sgemm	dense matrix-matrix multiplication	`SGM`	Algorithm	4	7/9/12/8	33/53/41/46	512B	X-P	7/9/8/8	[47]
hotspot	Estimate processor temperature	`HS`	Algorithm	8	3/5/6/6	35/38/36/38	3072B	Y-P	3/5/6/6	[46]
syrk	Symmetric rank-k operations	`SYK`	Cache-line	8	5/8/8/8	21/26/21/28	0	X-P	3/2/8/8	[48]
syr2k	Symmetric rank-2k operations	`S2K`	Cache line	8	6/6/8/8	33/38/33/19	0	X-P	1/1/6/6	[48]
atax	Matrix transpose and vector multiply	`ATX`	Cache line	8	6/8/8/8	13/17/17/22	0	X-P	1/1/1/1	[48]
mvt	Matrix vector product and transpose	`MVT`	Cache line	8	6/8/8/8	13/17/17/22	0	X-P	1/1/1/1	[48]
nbody	All-pairs gravitational n-body simulation	`NBO`	Cache line	8	2/4/6/6	24/38/35/46	0	Y-P	2/4/5/2	[40]
3DCONV	3D convention	`3CV`	Cache line	8	6/8/8/8	18/9/18/19	0	Y-P	6/8/8/8	[40]
bicg	BiCGStab linear solver	`BC`	Cache line	8	6/8/8/8	13/16/17/22	0	X-P	1/1/1/8	[48]
histogram	64-bin histogramming	`HST`	Data	8	6/8/8/8	15/19/20/15	1024B	X-P	5/5/6/7	[40]
B+tree	B+tree operations	`BTR`	Data	8	5/8/8/8	22/27/29/30	0	X-P	5/8/8/8	[46]
nw	DNA sequence alignment algorithm	`NW`	Writing	1	8/16/32/32	28/27/39/40	2180B	X-P	8/16/16/8	[46]
bfs	Breadth first search	`BFS`	Data&Writing	8	6/8/8/8	17/18/19/20	0	X-P	2/6/6/7	[46]
MonteCarlo	Option call price via MonteCarlo method	`MON`	Streaming	8	4/4/8/8	28/28/28/28	4096B	X-P	4/4/8/8	[40]
dxtc	High quality DXT compression	`DXT`	Streaming	2	8/8/10/10	63/89/89/91	2048B	X-P	8/8/10/10	[40]
sad	Sum of abs differences in MPEG encoder	`SAD`	Streaming	2	8/16/20/20	43/44/46/40	0	X-P	8/16/20/20	[47]
BlackScholes	Black-Scholes option pricing	`BS`	Streaming	4	8/16/16/16	23/25/21/19	0	X-P	8/16/16/12	[40]

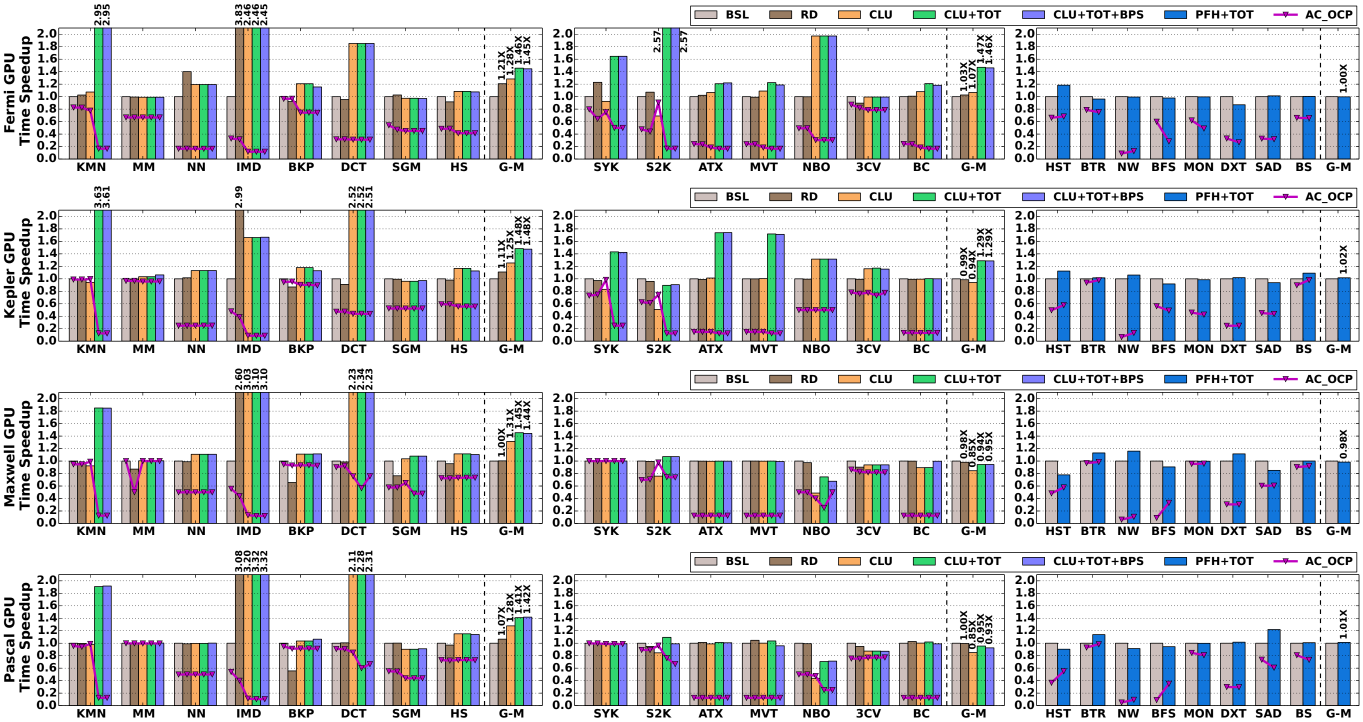


Figure 12: Normalized performance speedup and Achieved Occupancy (AC_OCP) obtained by CTA-Clustering and its associated optimizations on four GPU architectures. Three subfigures in each row include: (left) algorithm-related, (mid) cache-line related and (right) data, write and streaming related. "BSL" is the baseline. "RD" represents redirection-based clustering. "CLU" represents agent-based clustering with maximum allowable agents per SM as active agents. "TOT" stands for CTA throttling. "CLU+TOT" uses the optimal number of active agents per SM through throttling. "BPS" is cache bypassing. "PFH" represents CTA prefetching. Achieved Occupancy is defined as the ratio of the average active warps per active cycle to the maximum number of warps supported on an SM. It is not the theoretical Occupancy. All the results are normalized to the baseline and measured by the average of multiple runs.

(1) Although being quite effective for some applications (e.g., `NN`, `IMD`), the Redirection-based Clustering (**RD**) is not generally beneficial due to preassumed scheduling policy. Comparatively, the Agent-based techniques (e.g., CLU and CLU+TOT) are much more effective, especially for their low-cost SM-based binding (Section 4.2.3-(B)) on Fermi and Kepler due to their static CTA to warp-slot binding.

(2) Across architectures, CTA-Clustering appears to be more effective for algorithm-related applications as their inter-CTA reuse is inherent in algorithm itself. For cache-line related applications, CTA-Clustering only benefits Fermi

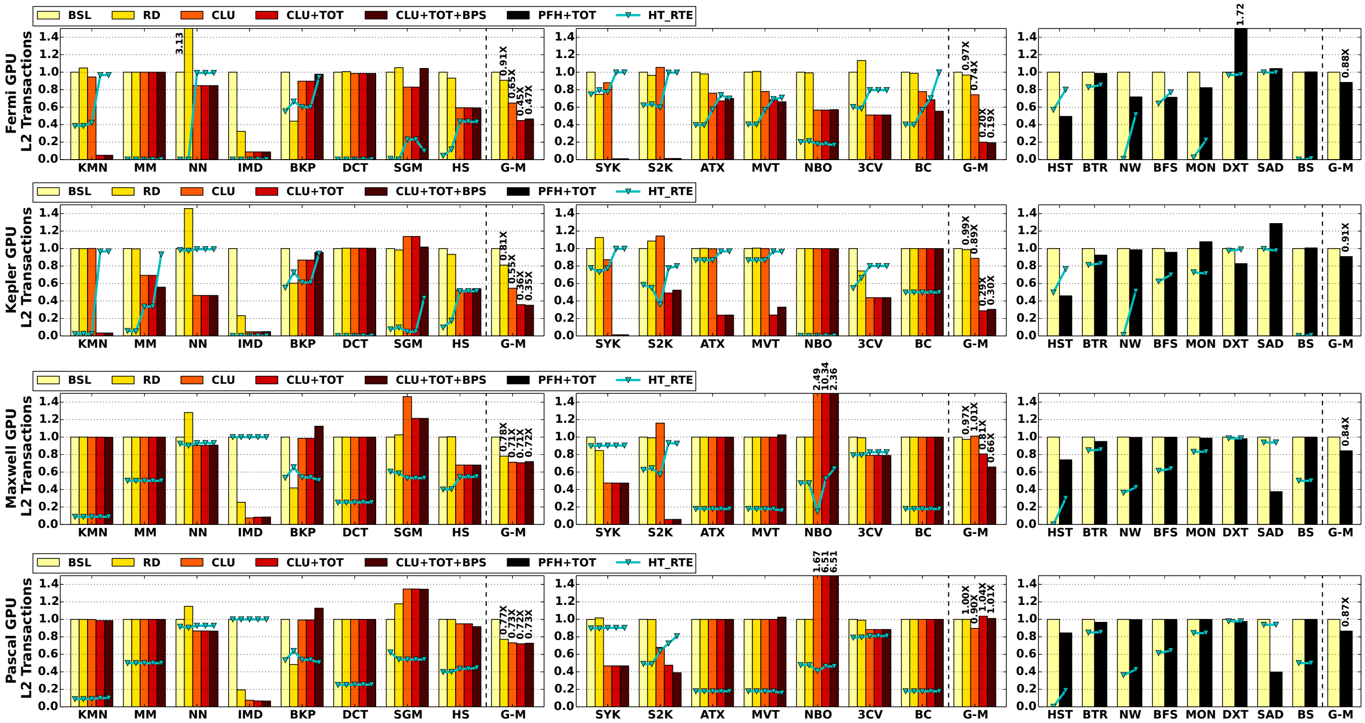


Figure 13: L2 cache transactions (or L1/Tex-L2 transactions) and L1 cache hit rates for CTA-Clustering and optimizations on Fermi, Kepler, Maxwell and Pascal GPUs.

and Kepler. This is because their L1 cache-line size is much larger than that of Maxwell and Pascal (128B vs. 32B). A larger cache-line implies more data fetching per miss, which facilitates possible spatial reuse across CTA boundaries. For Maxwell and Pascal, the 32B cache line is just one fourth of a load of a warp (4B×8), hence hardly any inter-CTA reuse. Clustering CTAs on these platforms for cache-line related applications may only introduce extra overhead.

(3) Regarding the three optimizations, throttling on CTA-Clustering is quite effective, especially for applications having high contention in execution resources (e.g., `KMN`, `SYK`, `S2K`, `ATX`, `MVT`). As a complementary optimization to CTA-Clustering and throttling, bypassing is in general ineffective because CTA throttling has already mitigated majority of the resource contention. For applications with no exploitable inter-CTA locality, CTA-Prefetching also gains overall insignificant performance improvement. There are two main reasons. (a) For cache-friendly GPU applications with large amount of access to L1 but under very limited cache capacity, prefetching may break the original locality and increase early evictions. (b) CTA prefetching requires repeated address and offset calculation which sometimes is quite complicated, incurring significant overhead. We want to emphasize that improving the performance for applications without exploitable inter-CTA locality is not the focus of this work. Our original goal for this basic scheme was to utilize the feature of imposed order from CTA-Clustering and provide a more general approach for addressing all GPU applications.

(4) Most of the algorithm-related cases (except for KMN) can benefit directly from clustering without CTA throttling to achieve the optimal performance (max_allowable_CTAs). Furthermore, we observed that the throttling-only approaches through controlling dynamic shared memory usage per CTA sometimes show far worse performance than our clustering-centric approach. This indicates that CTA throttling is not only unnecessary for some applications but also can be detrimental to performance sometimes. Thus it is complementary to CTA-Clustering and used only when capacity-conflict is constraining performance.

(5) Comparing Figure 13 with Figure 12, we confirm that L2 cache transaction is a better metric for evaluating overall GPU performance. In general, when the L2 transactions decline, the overall performance improves (e.g., `NBO` on Kepler), and vice versa (e.g., `NBO` on Maxwell and Pascal).

(6) Although we use `MM` as an example to explain the clustering process, its results are not very encouraging. This is not because `MM` does not have significant inter-CTA reuse, but rather due to three reasons: (1) The inter-CTA data reuse distance (e.g., the *S*-region in Figure 8-(A)) greatly surpasses the cache capacity. Several integer data per thread is already sufficient to saturate the at most 48KB L1 cache. However, the matrix size is usually larger than 1K by 1K. Thus an agent can hardly reuse any data temporally when starting a new task. One promising solution to reduce the reuse distance is to adopt the Tile-wise CTA indexing method shown in Figure 7. However, although we observed increase in hit rate and reduction in L2 transactions, its complex indexing calculation leads to significant overhead, bring little performance benefit. (2) While temporal locality is hard to leverage, spatial inter-CTA locality in `MM` is also difficult to exploit. In this MM implementation, there are 32 warps per CTA, severely limiting the number of the max allowable agents assigned per SM as well as the inter-agent data reuse. For instance, there is only one agent allocated per SM on Fermi, while the other three platforms have two. As discussed in Section 3.1, the L1/Tex unified caches on Maxwell and Pascal have two sectors and data sharing is not feasible among agents on two sectors. This is why little reuse and almost no performance gain and L2 transaction reduction are observed on Maxwell and Pascal.

Overall, CTA-Clustering shows slightly better performance on Fermi and Kepler than on Maxwell and Pascal because (i) Fermi and Kepler have a much larger L1 cache line, which is vital to spatial data reuse; (ii) Maxwell and Pascal need to endure the atomic and synchronization overhead for SM-based binding due to their dynamic CTA to warp-slots mapping; and (iii) Maxwell and Pascal L1/Tex unified cache is sectored, which can hinder cross-sector data reuse among agents.

6. Related Work

GPU Cache Performance. Existing proposals for improving GPU cache efficiency include thread throttling [31, 51, 52], cache bypassing [12–16, 19, 20], cache line utilization optimization [17, 18], replacement methods [35, 53] and prefetching [32, 36, 44]. However, when discussing data reuse, these existing proposals have neither demonstrated and quantified the existence of the inter-CTA locality on real GPU hardware, nor they have answered whether GPU hardware can exploit such reuse at L1. Additionally, majority of these studies require architecture modification through simulation which cannot be immediately deployed on existing GPUs. As for prefetching, current works [32, 36, 44] focus on providing intra-CTA solutions such as intra-warp and inter-warp stride prefetching.

CTA Scheduling. Previous works [8, 10, 11] have focused on designing efficient warp-level schedulers (i.e., intra-CTA approaches), e.g., achieving better off-chip memory bandwidth [8] or DRAM performance [11]. A few hardware approaches [22, 27, 31, 33, 36] have been proposed to improve the default CTA scheduler for efficient scheduling and better performance. Although these works provide useful insights on CTA scheduler design, they require to modify the default CTA scheduler and are validated through simulators. For example, following the common assumption that inter-CTA locality should be exploited at the shared L2 level, [22] proposed to redirect CTA index for exploiting inter-CTA locality on L2. It distributes CTAs among as many SMs as possible in a short interval to leverage the reuse in L2; while our approach is to cluster CTAs with inter-CTA locality to the same SM for on-chip L1 local reuse. Note that their redirection does not need to take care of the exact mapping from a CTA to an SM as the L2 is uniformly shared among SMs. To overcome the reality that the hardware CTA scheduler is unknown and inaccessible on commercial GPUs, our approach takes efforts to handle such mapping. [27, 36] modified the CTA scheduler to simply assign consecutive CTAs to the same SM to explore locality. They assumed the default CTA scheduler as round-robin and only considered a simple one-dimensional assignment scenario. More importantly, they did not demonstrate whether the inter-CTA locality can be harvested, either spatially or temporally, on L1 and how to bring instant performance benefits. Note that [27, 36] primarily focus on how many number of CTAs should be scheduled to an SM instead of inter-CTA locality. LaPerm [33] improves the parent-child CTA locality in irregular applications by modifying the hardware CTA-scheduler to enable scheduling strategies such as prioritizing the execution of the child CTAs and binds them to the SMs occupied by their parents. We propose a practical software-based solution that is general to both regular and irregular applications with the focus of exploiting inter-CTA locality on on-chip SM-private caches. LaPerm can be integrated into our framework to address data-related applications. Note that our proposed techniques are also generally orthogonal to the intra-warp and inter-warp optimization schemes [16, 34–36].

7. Conclusion

In this paper, we proposed a novel clustering technique for tapping into the performance potential of a largely ignored type of locality: inter-CTA locality. We first demonstrated the capability of the existing GPU hardware to exploit such locality, both spatially and temporally, on L1 or L1/Tex unified cache. To verify the potential of this locality, we quantified its existence in a broad spectrum of applications and discussed its sources of origin. Based on these insights, we proposed the concept of CTA-Clustering and its associated software techniques. Finally, we evaluated these techniques on all modern generations of NVIDIA GPU architectures. The experimental results showed that our proposed clustering techniques could significantly boost on-chip cache performance, resulting in substantial overall performance improvement.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments and suggestions for improving this work. This research is supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE - Center for Advanced Architecture Evaluation". The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830. This research is also supported by the HiPEAC Collaboration Grants.

References

[1] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[2] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures a dependence-based approach*. Morgan Kaufmann, 2001.

[3] Jingling Xue. *Loop tiling for parallelism*, volume 575. Springer Science & Business Media, 2012.

[4] James Philbin, Jan Edler, Otto J Anshus, Craig C Douglas, and Kai Li. Thread scheduling for cache locality. In *ACM SIGOPS Operating Systems Review*, volume 30, pages 60–71. ACM, 1996.

[5] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 47–58. ACM, 2007.

[6] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[7] NVIDIA. CUDA Programming Guide, 2015.

[8] Mengjie Mao, Wujie Wen, Xiaoxiao Liu, Jingtong Hu, Danghui Wang, Yiran Chen, and Hai Li. TEMP: thread batch enabled memory partitioning for GPU. In *Proceedings of the 53rd Annual Design Automation Conference*, page 65. ACM, 2016.

[9] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 235–246. ACM, 2011.

[10] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, pages 407–420. IEEE Computer Society, 2007.

[11] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 395–406. ACM, 2013.

[12] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. Adaptive and transparent cache bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 17. ACM, 2015.

[13] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. Locality-driven dynamic GPU cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 67–77. ACM, 2015.

[14] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive Cache Management for Energy-Efficient GPU Computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 343–355. IEEE Computer Society, 2014.

[15] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 516–523. IEEE Press, 2013.

[16] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 25–38, Oct 2015.

[17] Lingda Li, Ari B Hayes, Shuaiwen Leon Song, and Eddy Z Zhang. Tag-Split Cache for Efficient GPGPU Cache Utilization. In *Proceedings of the 2016 International Conference on Supercomputing*, page 43. ACM, 2016.

[18] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 86–98. ACM, 2013.

[19] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 272–283. IEEE, 2014.

[20] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. Coordinated static and dynamic cache bypassing for GPUs. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 76–88. IEEE, 2015.

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

[22] Bo-Cheng Charles Lai, Hsien-Kai Kuo, and Jing-Yang Jou. A cache hierarchy aware thread mapping methodology for GPGPUs. *IEEE Transactions on Computers (TC)*, 64(4):884–898, 2015.

[23] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. *Comput. Syst*, 26:63–72, 2009.

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

[25] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*, pages 119–130. ACM, 2015.

[26] Kunal Gupta, Jeff A Stuart, and John D Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar)*, pages 1–14. IEEE, 2012.

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

[28] NVIDIA. GTX980 Whitepaper: Featuring Maxwell, the Most Advanced GPU Ever Made, 2014.

[29] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. Cache coherence for GPU architectures. In *Proceedings of the 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 578–590. IEEE, 2013.

[30] Sara S. Baghsorkhi, Isaac Gelado, Matthieu Delahaye, and Wen-Mei Hwu. Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2012.

[31] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. Neither more nor less: optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques (PACT)*, pages 157–166. IEEE Press, 2013.

[32] Hyeran Jeon, Gunjae Koo, and Murali Annavaram. CTA-aware Prefetching for GPGPU. *Computer Engineering Technical Report Number CENG-2014-08*, 2014.

[33] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 583–595. IEEE Press, 2016.

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

[35] A. Sethia, D. A. Jamshidi, and S. Mahlke. Mascar: Speeding up GPU warps by reducing memory pitstops. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 174–185, Feb 2015.

[36] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 332–343. ACM, 2013.

[37] Lingda Li, Ari B Hayes, Stephen A Hackler, Eddy Z Zhang, Mario Szegedy, and Shuaiwen Leon Song. A Graph-based Model for GPU Caching Problems. *arXiv preprint arXiv:1605.02043*, 2016.

[38] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 369–380. ACM, 2011.

[39] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. Hybrid CPU-GPU scheduling and execution of tree traversals. In *Proceedings of the 2016 International Conference on Supercomputing (ICS)*, page 2. ACM, 2016.

[40] NVIDIA. CUDA SDK Code Samples, 2015.

[41] Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[42] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. SFU-Driven Transparent Approximation Acceleration on GPUs. In *Proceedings of the International Conference on Supercomputing (ICS)*, page 15. ACM, 2016.

[43] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418. ACM, 2013.

[44] Jaekyu Lee, Nagesh B Lakshminarayana, Hyesoon Kim, and Richard Vuduc. Many-thread aware prefetching mechanisms for GPGPU applications. In *Proceedings of the 43rd Annual International Symposium on Microarchitecture (MICRO)*, pages 213–224. IEEE, 2010.

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

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

[47] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[48] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar)*. IEEE, 2012.

[49] NVIDIA. CUDA Profiler User's Guide, 2015.

[50] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing (ICS)*, pages 15–24. ACM, 2012.

[51] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83. IEEE Computer Society, 2012.

[52] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Divergence-aware Warp Scheduling. In *Proceedings of the 46th Annual International Symposium on Microarchitecture*, MICRO-46, pages 99–110. ACM, 2013.

[53] Jayesh Gaur, Raghuram Srinivasan, Sreenivas Subramoney, and Mainak Chaudhuri. Efficient Management of Last-level Caches in Graphics Processors for 3D Scene Rendering Workloads. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 395–407. ACM, 2013.