Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations

Bo Wu[◇], Guoyang Chen[⋆], Dong Li[+], Xipeng Shen[⋆], Jeffrey Vetter[†]
[◇]Department of Electrical Engineering and Computer Science, Colorado School of Mines, USA
[⋆]Department of Computer Science, North Carolina State University, USA
[+]Department of Computer Science, University of California, Merced, USA
[†]Oak Ridge National Laboratory, USA
[◇]bwu@mines.edu,[⋆]{gchen11,xshen5}@ncsu.edu,[+]dli35@ucmerced.edu,[†]vetter@ornl.gov

ABSTRACT

A GPU's computing power lies in its abundant memory bandwidth and massive parallelism. However, its hardware thread schedulers, despite being able to quickly distribute computation to processors, often fail to capitalize on program characteristics effectively, achieving only a fraction of the GPU's full potential. Moreover, current GPUs do not allow programmers or compilers to control this thread scheduling, forfeiting important optimization opportunities at the program level. This paper presents a transformation centered on Streaming Multiprocessors (SM); this software approach to circumventing the limitations of the hardware scheduler allows flexible program-level control of scheduling. By permitting precise control of job locality on SMs, the transformation overcomes inherent limitations in prior methods.

With this technique, flexible control of GPU scheduling at the program level becomes feasible, which opens up new opportunities for GPU program optimizations. The second part of the paper explores how the new opportunities could be leveraged for GPU performance enhancement, what complexities there are, and how to address them. We show that some simple optimization techniques can enhance co-runs of multiple kernels and improve data locality of irregular applications, producing 20-33% average increase in performance, system throughput, and average turnaround time.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*optimization, compilers*

General Terms

Performance, Experimentation

ICS'15, June 8–11, 2015, Newport Beach, CA, USA.

http://dx.doi.org/10.1145/2751205.2751213.

Keywords

GPGPU, Scheduling, Compiler Transformation, Data Affinity, Program Co-Run

1. INTRODUCTION

Recent years have seen increasing popularity of Graphics Processing Units (GPUs) in general-purpose computing, thanks to massive parallelism provided by GPUs. With hundreds of cores integrated, the GPU often creates tens of thousands of threads for an application. The massive parallelism produces large potential throughput, but also imposes grand challenges for thread management, or scheduling.

Scheduling determines when and where a task is processed. It is essential for matching communication and memory access patterns with underlying architecture, in order to fully tap into the power of a parallel system. Scheduling is usually controlled by thread schedulers. On CPU, the thread scheduling is implemented through system APIs. But on GPUs, there is no such software API; the scheduling on GPUs has been controlled by hardware and runtime. Such a design is demanded by the scale of parallelism: Hundreds of thousands of threads need to be scheduled in very short time. However, the lack of software-level control of scheduling forms a major barrier for software to leverage scheduling to optimize program executions. What increases the barrier is that the scheduling algorithms employed by GPU hardware and runtime have remained non-disclosed; the schedulers vary substantially across generations and have exhibited some obscure and non-deterministic behaviors (detailed in the next section).

The restrictions have drawn some recent attentions from researchers in various domains. A number of studies independently invented the method of *persistent threads* to go around the hardware scheduling problem [3,9,13,41,44]. The idea is to create only a small number of threads that can simultaneously run actively on a GPU. Unlike in traditional kernels where a thread terminates as it finishes a task, these threads stay alive throughout the execution of a kernel function. They continuously fetch and execute tasks from one or more task queues. By controlling the order of the tasks in the queues, one can match the executions with some communication patterns among tasks—for example, putting a producer and its consumer into the same queue.

Although persistent threads offers some support to task scheduling on the GPU, the support is restrictive. It only decides which tasks map to which persistent thread and their execution order; it gives no support for deciding *where or*

on which processor a task should run. Such location control is still up to the hardware and proprietary runtime, which decide the placement of persistent threads, and hence the placement of tasks associated with those threads.

Lack of such scheduling control at the spatial dimension hinders persistent threads in supporting optimizations that are related with non-uniformity in processors. For instance, a modern GPU consists of multiple streaming multiprocessors (SM), with each containing tens of cores. Cores on one SM usually share some on-chip storage on that SM (e.g., L1 cache and texture cache). As a result, one task may be able to read the data in a cache brought by another task that concurrently runs on the same SM. With location control, one could make two tasks that share lots of data run concurrently on the same SM[1]. Such optimizations are especially beneficial for tasks with non-uniform data sharing, which include tasks of many irregular applications (e.g., N-body simulations), as well as tasks coming from different kernels (or applications) that are deployed concurrently on a GPU. Besides for data sharing, the spatial control is critical when there are architectural variations among SMs. Unintentional variations among SMs in a GPU already widely exist today [16]; with frequency scaling [21] possibly implemented in future GPUs, even more substantial (intentional) variations (e.g., different SMs could be reconfigured to different clock frequencies to balance energy and performance) are possible. In these scenarios, spatial control of scheduling is important for matching tasks with the suitable SMs.

In this work, we show that spatial scheduling control actually can be enabled through a simple program transformation, called *SM-centric transformation*.

SM-centric transformation includes two essential techniques. The first is *SM-based task selection*. In a traditional GPU kernel execution, with or without persistent threads, what tasks a thread executes are usually based on the ID of the thread (or determined randomly in a dynamic task management). While with SM-based task selection, what tasks a thread executes is based on the ID of the SM that the thread runs on. By replacing the binding between tasks and threads with the binding between tasks and SMs, the scheme enables a direct, precise control of task placement on SM.

The second technique is *filling-retreating scheme*, which offers a flexible control of the amount of active threads on an SM. Importantly, the control is resilient to the randomness and obscuration in GPU hardware thread scheduling. It helps SM-centric transformation in two aspects. First, it ensures an even distribution of active threads on SMs, which is vital for guaranteeing the correctness of SM-centric transformations. Second, it facilitates online determination of the parallelism level suitable for a kernel, which is especially important for the performance of multiple-kernel co-runs, a scenario benefiting significantly from SM-centric transformation.

SM-centric transformation, by enabling flexible program-level control of task scheduling, opens up many new opportunities for optimizations. The second part of the paper explores how the new opportunities could be leveraged for GPU performance enhancement, what complexities there are, and how to address them. Specifically, we explore the use of the scheduling flexibility to enable affinity-oriented task scheduling and SM partition for co-runs of multiple kernels. In our experiments on 72 co-runs of kernels, it helps produce on average 33% improvement in system throughput and turnaround time. When applied to locality enhancement, the enabled spatial scheduling shortens the execution times of four irregular applications by 20% on average. In both cases, it significantly outperforms the support that persistent threads provide. These results indicate that SM-centric transformation, by complementing prior methods, provides a critical missing piece of the puzzle for enabling and exploiting flexible control of task scheduling on the GPU.

In summary, this work makes following contributions.

- It proposes SM-centric transformation, which, for the first time enables precise spatial scheduling of GPU tasks. It offers the missing piece of the puzzle for circumventing GPU hardware restrictions to implement a flexible control of task scheduling.
- It uncovers the potential of the enabled scheduling control for both single-kernel runs and multi-kernel co-runs.
- It reveals some major challenges for leveraging spatial scheduling on the GPU, and develops a set of practical solutions; experiments demonstrate their effectiveness and the tangible and substantial benefits with the proposed program-level schedule control.

In the rest of this paper, Section 2 provides some necessary background on GPU scheduling, Section 3 describes SM-centric transformation, Section 4 discusses the uses of the enabled spatial scheduling for both locality enhancement in single-kernel runs and throughput improvement in multi-kernel co-runs, and discusses some major challenges. Section 5 presents some simple solutions to those challenges, through which, we are able to examine the practical benefits of the spatial scheduling. Section 6 reports experimental results, followed by related work and conclusions.

2. BACKGROUND

We base our discussions on terms in NVIDIA CUDA [1]; but the technique could be applied to other GPU programming models.

Organization of Cores and Threads.

As a massively parallel architecture, a GPU consists of a number of *streaming multiprocessors (SM)*, with each containing tens of cores. A GPU usually creates a large number of threads at the launch of a *kernel* (i.e., a CPU-invoked function that runs on GPU). These threads are typically organized in a hierarchy: 32 compose a *warp*, many warps compose a *thread block* or called a *cooperative thread array (CTA)* and many CTAs compose a *grid*.

Spatial Scheduling.

A CTA is the unit for spatial scheduling: At a kernel launch, the GPU hardware scheduler named GigaThread [32] assigns each CTA to one of its SMs. The assignment algorithm has not been disclosed to the public. It differs from

[1]Mapping two tasks to the same persistent thread can also make them map to the same SM, but the tasks have to run serially by that thread, throttling the benefits of synergistic data fetching.

one generation of GPUs to another, and exhibits lots of irregularity. For example, our experiments on Tesla M2075, a type of widely used workstation GPUs, show different CTA-to-SM assignments in two repeated invocations of the same kernel on the same input, and neither is in a round-robin or other regular predictable pattern.

Temporal Scheduling.

A warp is the unit for temporal scheduling: All threads in a warp proceed in lockstep. Many CTAs may be assigned to an SM, but at one time point, only a limited number of them can be active—meaning that they attain enough registers and other hardware resources and are ready to run. All other CTAs have to wait until some active CTA finishes executing the entire kernel function and releases some hardware resources.

Non-Uniformity on GPU.

Spatial scheduling is potentially beneficial to the GPU, as non-uniformity exists on both GPU resource sharing and its workload.

On the resource sharing aspect, modern GPU features non-uniform cache sharing. In Tesla M2075, for instance, there are 14 SMs, with each containing some cache—such as, instruction cache, L1 data cache, constant cache, and texture cache—that is shared by all cores on that SM but is not accessible by other SMs.

On the workload aspect, non-uniformity shows in two levels. For a single GPU kernel, a CTA may share different amounts of data with different CTAs. Molecular Dynamics (MD) simulation is such an example. It simulates interactions among neighbor atoms. The atoms simulated by two CTAs may have many or few neighbors, depending on the distances among them in the simulated space. That naturally leads to non-uniform data sharing among CTAs. Meanwhile, recent generations of GPUs start to support concurrent executions of multiple kernels on a single GPU. Although currently the kernels have to be launched from a single CUDA context, a more general support for concurrent executions of multiple GPU applications is expected to come in the near future. In these co-run scenario's, non-uniformity becomes even more common: CTAs from the same kernel often share more instructions and data than CTAs from different kernels do.

The non-uniformity suggests the potential of spatial scheduling. As Section 6 quantitatively confirms, a good spatial scheduling may bring an over 30% speedup on average.

3. SM-CENTRIC TRANSFORMATION

At the center of SM-centric transformation are two techniques: SM-centric task selection, and a filling-retreating scheme. In this section, we first explain the basic ideas of the two techniques and how they complement each other to form a single solution to circumvent the limitation from the hardware scheduler. As the techniques are generally applicable to various GPU programming models, we use high-level pseudo-code for description and skip detailed complexities in implementation so that the ideas can be easily grasped by general readers. We then use CUDA as an example programming model to explain the detailed implementation of the techniques, including some subtle considerations that are critical for the techniques to work efficiently. We show that the entire SM-centric transformation can be conducted through a simple pass by compilers. At the end, we give some discussions on the soundness of the transformation and its applicable conditions.

3.1 SM-Centric Task Selection

Basic Idea.

SM-centric task selection associates tasks with SMs. We explain it based on the following abstract model of GPU kernel executions.

Commonly, an invocation of a GPU kernel causes many GPU threads to create, which are often organized in a hierarchical structure. At an abstract level, the execution of a GPU kernel can be regarded as consisting of many jobs[2] conducted in parallel by a number of workers on some SMs. Here, a worker corresponds to a group of GPU threads (e.g., a CTA), and a job corresponds to the operations conducted by such a thread group, including all their data accesses. There is a unique ID number associated with each job, worker, and SM.

In traditional GPU programs, which job a worker does has been determined by the worker's ID, as the pseudo code in Figure 1 (a) shows. The technique of *Persistent threads* maps multiple jobs to a single worker, but the set of jobs for a worker is still determined by the worker's ID, as shown in Figure 1 (b). Since the placement of workers on SMs is controlled by the hardware schedulers, the job-worker binding makes the placement of jobs on SMs solely depend on the hardware schedulers.

The idea behind SM-centric task selection is to replace the job-worker binding with a binding established between jobs and SMs. As Figure 1 (c) shows, a job queue is built for every SM before the invocation of a kernel function. Inside the kernel function, each worker first figures out on what SM it resides, and then uses the SM ID to fetch the next job in the corresponding job queue to execute. In this way, controlling the placement of a job on a specific SM becomes simple: Just putting that job's ID into the job queue of that SM.

The idea is straightforward. But some complexities must be addressed to implement the idea soundly and efficiently.

Correctness Issues by Hardware Schedulers.

Through a close look at the pseudocode in Figure 1 (c), one will see that for it to work correctly on a GPU program, the number of workers assigned to an SM must be no fewer than the number of jobs assigned to the SM. It is because in that code, one worker on an SM processes only one job assigned to that SM. Some jobs on that SM would be left unprocessed if the number of workers is less than the number of jobs.

However, how many workers are assigned to an SM is determined by the GPU hardware scheduler. Our experiments indicate that the assignment by hardware schedulers is often unpredictable. On a Tesla M2075 with 14 SMs, for instance, when running a kernel with 1400 workers (i.e., CTAs), we observe an uneven distribution of workers: the number of workers per SM varies from 92 to 110. And when running the kernel with 14 workers, some SMs get multiple workers

[2]In this paper, "job" and "task" are interchangeable terms, although we tend to use "job" more often when referring to entities in this abstract kernel execution model.

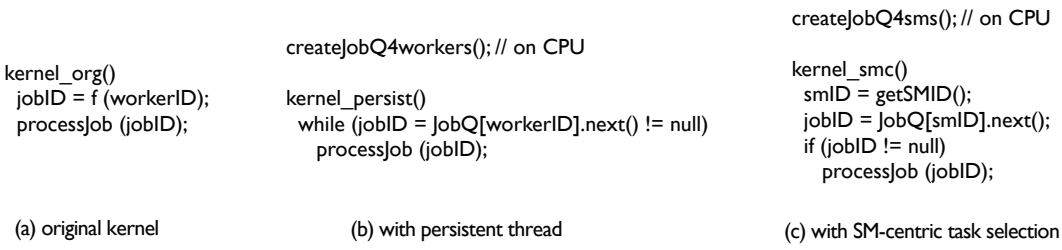

Figure 1: Conceptual relations among jobs, workers, and SMs.

while others get none. Moreover, the worker distribution varies from run to run, displaying lots of randomness.

Such non-determinism jeopardizes the soundness of the basic SM-centric task selection. An option is to allow dynamic job stealing such that workers on one SM can steal jobs left on another SM. It requires more complicated code to be inserted into the GPU kernel to implement the job stealing logic, and hence increases register pressure and reduces parallelism. More importantly, the stealing changes the intended job-to-SM mapping.

3.2 Filling-Retreating Scheme

We address the complexity through a *filling-retreating scheme*. This scheme offers a simple way to precisely control the number of active workers on each SM.

The scheme works hand-in-hand with the concept of persistent threads. Similar to persistent threads, with this scheme, a small number of workers are kept alive for each SM throughout the kernel execution. These workers continuously fetch and process the jobs assigned to the SM until the queue gets empty. The tricky part is on how to precisely control the number of active persistent threads (or in our term, active workers) for each SM.

Filling-retreating offers a simple solution. It leverages a common property of GPU schedulers. On GPUs, each SM can only support a limited number of active workers at the same time due to hardware limitation. On all GPUs we tested, despite the differences in their schedulers, one common property is that they always try to assign a worker to an SM that can still accommodate some active workers if there is any (rather than putting the worker into a waiting queue of an SM).

Suppose that one SM can support at most m active workers at the same time. In the filling-retreating scheme, a total of $m * M$ workers are created at a kernel launch, where M is the number of SMs in the GPU. Due to the aforementioned common property, each SM gets m workers assigned. This step is the "filling" part of the scheme.

Although the "filling" step ensures every SM gets m workers, as multiple studies have shown [20,25], having the largest number of workers on an SM is not always the best for maximizing the computing efficiency due to cache and bus contention. This phenomenon is also confirmed in Section 6. The "retreating" part of our scheme facilitates flexible adjustment of the number of active workers on an SM. Suppose one wants to have n_{target} active workers per SM. A counter is created for each SM to record the number of workers that have started processing jobs on that SM. Each worker, before starting working on a job, first atomically increases the corresponding counter and then checks whether the counter

```
createJobQ4sms(); // on CPU

kernel_smc()
  smID = getSMID();
  workers = workerCounters[smID]++; // atomic
  if (workers > wantedNumPerSM)
    return;
  while (jobID = JobQ[smID].next() != null)
    processJob (jobID);
```

Figure 2: Psuedo code of a GPU kernel in a *filling-retreating* scheme.

value already exceeds n_{target}. If so, the worker exits immediately. Figure 2 shows the pseudo code.

The correctness of the filling-retreating scheme relies on the fast distribution of thread blocks by the hardware scheduler. That is, the filling phase should finish before any thread block retreats (i.e., exiting its execution). Otherwise, the hardware scheduler could assign totally more than m workers onto an SM because of the vacancy on that SM formed by the early retreat of some workers on it. Correspondingly, some other SMs would get less than m workers assigned. Fortunately, our experiments show that such cases have never happened, plausibly due to the extreme speed of the hardware-based assignment of workers. A check put into the runtime driver could further ensure the condition to hold, the necessity of which is not shown in our experiments.

The benefits of the precise control of the number of active workers on an SM goes beyond helping with the correctness of SM-centric task selection. It also enables a precise control of parallelism on GPU, which facilitates flexible partitions of SMs among co-running kernels shown in Section 5.

3.3 Implementation

The SM-centric transformation can be easily applied either manually or through a compiler. For proof of concept, we build a prototype source-to-source compiler based on Cetus [22], in which the transformation is implemented as a pass over the input code. The experience taught us the importance of several subtle considerations in the design, which we highlight next before showing the full details of the implementation.

First, the dequeue operation in the while loop in Figure 2 is good for illustrating the basic idea but poor for performance. An atomic operation could cause substantial overhead especially when the work in the loop body is small. When implementing the transformation, it is important to avoid such atomic operations in the while loop.

In our design, we circumvent the needs for atomic job fetching by leveraging a property offered by the filling-retreating scheme: There are precisely N_{target} active workers on an SM. With that property, each active worker only needs to process W/N_{target} jobs, where W is the total number of jobs assigned to the SM. So if we put the job IDs of an SM into an array, the set of jobs for a worker just corresponds to a segment of the array. The starting and ending indices of the segment can be easily attained before the worker enters the job fetching and processing loop. With this improvement, the while loop in Figure 2 can be converted into a simple for loop, iterating the elements in the segment assigned to the worker, and the atomic operation can be hence removed from the loop. In our implementation, we actually use a single array to store the IDs of all jobs. The set of jobs of an SM corresponds to just one section of the array. The position of a job ID in the array hence determines on which SM it will be processed. (Lines 9 to 16 in Listing 1 implement this design; explained later.)

Second, the ID of an SM can be obtained efficiently. CUDA, like the C programming language, allows programmers to insert assembly code, which is designed by NVIDIA as an intermediate representation named Parallel Thread Execution (PTX) [34]. It has a special register, %smid, which stores the SM identifier. One "mov" instruction can copy the value in %smid to an integer variable. Line 27 in Listing 1 shows the code.

Listing 1: Macros that materialize SM-centric transformation (N jobs; M SMs).

```
1  #define  __SMC_init  \
2  unsigned int * __SMC_workersNeeded = __SMC_numNeeded(); \
3  unsigned int * __SMC_newChunkSeq = __SMC_buildChunkSeq(); \
4  unsigned int * __SMC_workerCount= __SMC_initiateArray();
5
6  #define  __SMC_Begin  \
7  __shared int __SMC_workingCTAs; \
8  __SMC_getSMid; \
9  if(offsetInCTA == 0) \
10   __SMC_workingCTAs = \
11     atomicInc (&__SMC_workerCount[__SMC_smid], INT_MAX); \
12 synchthreads(); \
13 if(__SMCS_workingCTAs >= __SMC_workersNeeded) return; \
14 int __SMC_chunksPerCTA = \
15     __SMC_chunksPerSM / __SMC_workersNeeded; \
16 int __SMC_startChunkIDidx = __SMC_smid * __SMC_chunksPerSM + \
17     __SMC_workingCTAs * __SMC_chunksPerCTA;\
18 for (int __SMC_chunkIDidx = __SMC_startChunkIDidx; \
19     __SMC_chunkIDidx < __SMC_startChunkIDidx+\
20                        __SMC_chunksPerCTA ; \
21     __SMC_chunkIDidx++) { \
22     __SMC_chunkID = __SMC_newChunkSeq[__SMC_chunkIDidx];
23
24 #define  __SMC_End  }
25
26 // get the ID of the current SM
27 #define  __SMC_getSMid  \
28 uint __SMC_smid;\
29 asm("mov.u32 %0, %smid;" : "=r"(__SMC_smid) )
```

Details.

In this part, we describe some low-level complexities that our description has skipped. The discussion is based on CUDA, but the implementation can be done for other GPU programming models, such as OpenCL.

On GPU, the spatial scheduling unit is not a thread but a CTA, an array of threads. Correspondingly, the job assignments to processors in our design is in the unit of *job chunks*—the set of jobs executed by a CTA in the original GPU program. In a typical GPU program, the thread ID is used to distinguish jobs, and one CTA handles one job chunk; the ID of a CTA in the original program is hence treated as the ID of the job chunk that CTA processes.

To minimize changes needed to the original GPU program, we encapsulate most parts of the code for SM-centric transformation into four macros. With them, applying SM-centric transformation involves only several minor changes to the original GPU program. As Listing 2 shows, on the CPU-side code, it inserts one macro, *__SMC_init*, before the invocation of a GPU kernel, and appends three arguments to the kernel call. On the GPU-side code, it inserts the calls to two other macros, *__SMC_Begin* and *__SMC_End*, and replaces the appearances of the ID of CTA in the kernel with *__SMC_chunkID*. These can be done easily by the compiler in one pass over the original GPU program.

The above four macros are defined in Listing 1. The first, *__SMC_init*, initiates the three variables with the number of workers needed, the array of the desired sequence of IDs of job chunks, and an all-zero counter array to count active workers. The functions used to initiate the first two variables can be provided by the programmer or the optimizing compiler; their definitions depend on the purpose of the specific application of the SM-centric transformation, as Section 5 will illustrate. The second macro, *__SMC_Begin*, first calls the fourth macro to get the ID of the SM by reading the particular register, then checks whether the SM already has enough active CTAs. If not, it computes the starting and ending positions of the sets of jobs it should work on, gets into the for loop to process them one by one. The third macro, *__SMC_End*, is trivial, just putting in the ending bracket of the "for" loop in the second macro.

3.4 Soundness

At a high level, SM-centric transformation manipulates the association between jobs and processors, and hence alters the mapping between jobs and threads and possibly the execution order of the jobs. As a kind of remapping transformation as persistent threads is, for SM-centric transformation to work soundly, the GPU program needs to meet the same conditions as in the case of persistent threads [9, 13]:

(1) The operations by different threads are discriminated only by the global thread ID; (2) The execution order of the CTAs does not disturb the correctness of the kernel.

The first condition ensures that SM-centric transformation does not change integrity of a job even though *all* appearances of the CTA ID in a kernel are replaced with the *__SMC_chunkId*. We note that even though current GPUs do not migrate CTAs across SMs, the job integrity holds even if CTA migrates—given that the attainment of *__SMC_chunkId* is atomic. The second condition ensures that the new order of execution maintains the meaning of the program.

The two conditions hold for well-formed GPU programs, due to the nature of GPU execution models. At a high level, they are Single-Program-Multiple-Data (SPMD) models; all GPU threads at a kernel launch execute the same function, while their specific operations are determined only by the thread ID. Meanwhile, for a GPU program to work properly,

Listing 2: GPU program after SM-centric transformation (N vector elements; K job chunks; M SMs).

```
1  /**** CPU−side code ****/
2  main (){
3    ...
4    __SMC_init;
5    invoke original kernel with three extra arguments:
6          __SMC_chunkCount, __SMC_newChunkSeq, K/M    .
7    ...
8  }
9
10 /**** GPU−side code ****/
11 __global__ kernel (...,
12 unsigned int *__SMC_chunkCount,
13 unsigned int *__SMC_newChunkSeq,
14 unsigned int __SMC_chunksPerSM)
15 {
16   __SMC_Begin
17   // the original kernel with the ID of CTA
18   // replaced with __SMC_chunkID
19   ... ...
20   __SMC_End
21 }
```

it should not rely on the execution order of CTAs, because due to the non-determinism in CTA scheduling on the GPU, it is hard to know what order would be taken in a run. Free from data race helps ensure the conditions hold. Recent years have seen a number of studies on data race detection for GPUs [5,47], which could serve as part of the automatic check of the applicability of SM-centric transformation.

4. USES AND COMPLEXITIES

By enabling program-level spatial scheduling, SM-centric transformation opens up some new opportunities for GPU optimizations. This section discusses some of them, and examines the main complexities associated with these new opportunities.

4.1 Example Uses

SM Partition for Multi-kernel Co-runs.

It has been observed that many GPU kernels exhibit sublinear speedups when the number of SMs used for the kernel increases [2, 33]. As a result, simulations have shown that if the set of SMs in a GPU can be partitioned such that different subsets of SMs work for different kernels concurrently, the system often gives higher throughput and the kernels manifest better overall responsiveness [2]. However, such partitions have not been feasible in practice for lack of scheduler controllability. On NVIDIA GPUs, for instance, when two kernels are launched concurrently (each usually has many CTAs), their CTAs are assigned to all SMs. And if the threads by one kernel already use too much register or shared memory on an SM, before its completion, the other kernel cannot start, hence resulting in a serial execution of the two kernels.

With minor extension to the SM-centric transformation, partitions of SMs among concurrent kernels becomes possible. For instance, if we want the first 6 SMs to work for kernel f_1, and the remaining 8 SMs for kernel f_2, we can set the mapping array used in SM-centric transformation such that all jobs of f_1 map to the first 6 SMs and those of f_2 to the other 8 SMs. When the two kernels get launched, the GPU scheduler still assigns CTAs of both kernels to every SM. However, a statement is inserted in each kernel after obtaining the SM ID, which checks whether the ID of this SM is one of the SMs supposed to work for this kernel. If not, the CTA returns immediately so that the SM can work for the other kernel.

Affinity-Based Scheduling.

Many GPU applications have inherent non-uniform data interactions, such as the MD example mentioned in Section 2. It causes non-uniform data sharing among job chunks. Following the concepts on traditional CPU [49], we state that two job chunks have good reference affinity if they share lots of data. As Section 2 mentions, each SM has an on-chip cache. So, if we can manage to assign onto the same SM the job chunks with good affinity, we may enhance the performance of the cache. SM-centric transformation makes this affinity-based scheduling possible.

4.2 Complexities

There are some complexities for applying SM-centric transformation to the use cases mentioned earlier.

For SM partition for co-runs, the key is to decide the best partition. For affinity-based scheduling, the key is to compute the affinity among job chunks and then group job chunks accordingly.

Additionally, there is a common complexity existed in both use cases: determining the suitable number of active CTAs for a kernel. As some studies have shown [20], creating the maximum number of CTAs that an SM can hold often gives suboptimal performance, because of cache and bus contention among them. The CTA aggregation employed in SM-centric transformation allows flexible control of the number of active CTAs for a kernel.

However, determining the suitable numbers of active CTAs is challenging. It depends on many factors, including the interaction between SM partitioning and data locality, program inputs, kernels' resource requirement and so on. Moreover, when coupling with the various methods to partition SMs (for co-runs), they could result in a large search space. For 2 kernels on a 14-SM GPU, if an SM can support at most 6 active CTAs, the search space contains $6 \times 6 \times 14 = 504$ cases.

5. DESIGNS FOR VALIDATION

In this section, we describe our design to address the complexities listed in the previous section. Our goal is to validate the practical value of the spatial scheduling enabled by SM-centric transformation, rather than to find the best solution to those complexities. Simplicity and practicality are the principles in our design. The rationale is that if the enabled spatial scheduling could bring substantial benefits with minimum support, the promise of the technique is validated.

5.1 Optimal Configuration Search

We first discuss the challenges for determining the best number of active CTAs (i.e., the n_{target} mentioned in Section 3.2, which is also called parallelism control) for a kernel and for finding the best partition of SMs between co-running kernels. We call these parameters together as a configura-

tion in our discussion. The difficulty is that the space of the configuration values is large and the best configuration depends on many factors. It is often too costly to try every possible configuration at runtime. We employ the standard sampling method to efficiently approximate the best configuration. When a kernel is inside a loop, the sampling may happen during the first several iterations; otherwise, the sampling may happen offline or across runs.

Because SM partition mainly affects interactions across SMs, while the parallelism control is mainly related with resource usage inside an SM, we observe that the optimal level of parallelism for a kernel is only loosely connected with how we partition SMs. Hence our search scheme first evenly partitions the SMs among kernels, and tries to find the appropriate numbers of active CTAs for each kernel. It starts with the maximum CTAs supported by an SM for the kernel (no larger than 8), and decreases the number by 1 in each iteration until it observes decreased performance or the number reaches 1. This is a typical process of hill climbing. After that, our search scheme fixes the CTA numbers but adjusts SM partition by setting the number of SMs assigned to a co-run kernel from 1 to the maximum-1 (in a step size of three) while the rest SMs are used for the other co-running kernel. As a prior study [33] does, this work considers only co-runs of two kernels. Based on the sampled data, the optimal partition is approximated through interpolation.

Like other online sampling-based approaches, our search scheme cannot work well when different iterations behave dramatically differently. Combining the sampling approach with domain knowledge about the behavior patterns of the program may help, which is out of the scope of this paper. In our experiment, we encountered only one such program, *reduction*. We did not give special treatment to it. The results in Section 6 show that even though the sampling method finds only suboptimal configurations for some programs, the overall benefits are still substantial, confirming the value of the SM-centric transformation.

5.2 Affinity-Based Scheduling

To implement the affinity-based scheduling, we model the scheduling problem as a graph partitioning problem. The modeling consists of two steps: graph construction and graph partitioning.

Graph Construction.

This step establishes a set of graphs named *affinity graphs*, in which, each vertex represents a job chunk and each edge weight represents the affinity score between two job chunks. Affinity score is defined as follows. Let S_1 and S_2 be the set of data blocks assessed by two job chunks J_1 and J_2 respectively. Their affinity score is $\frac{|S1 \cap S2|}{|S1 \cup S2|}$. There is no edge between two vertices when their affinity score is less than a threshold (0.05 in our experiment). If the affinity score is too small, there is only a small amount of data sharing and its effects on performance is negligible; ignoring them often breaks one affinity graph into multiple smaller graphs, allowing efficient graph partitioning in the next step.

Compiler techniques exist for analyzing working sets [35] for regular applications. On irregular applications, it is challenging as data access patterns may be unknown until run time. Runtime inspection techniques have been proposed to analyze data access patterns [27]. For GPU programs, prior work has shown the feasibility to employ CPU to implement the inspection asynchronously when the GPU is executing the kernel [45]. In this work, we used simple synchronous parallel inspection for irregular programs. But the asynchronous method may further reduce the overhead.

Graph Partitioning.

Given M SMs, this step partitions the set of vertices into M equal-size clusters. The job chunks corresponding to a cluster are scheduled to one single SM. This problem is known to be NP-hard. There are some existing heuristic algorithms, but we find them costly. Instead, we design a random, lightweight and highly parallelizable algorithm. Its basic idea is to select a seed vertex for each cluster and greedily enlarge each cluster to include the vertices that have high affinity scores with the selected vertices. The algorithm has three steps. (1) *Seeds selection.* Selecting the seed vertices is important for the partitioning quality; we try to minimize the affinity among them. The initial seed set is formed by randomly selecting a vertex from each of the affinity graphs. If the number of affinity graphs is no less than M, only M of them are randomly selected. If there are less than M seeds in the set, we iterate over the remaining vertices until we find one, whose affinity scores with all current seeds are smaller than a threshold (initialized to 0) and add it to the seed set. This step stops once we get M seeds. After iterating all vertices if we still need more seeds, we increase the threshold by 0.1 and start the next round of search. Ten rounds are needed at most as the threshold would grow to 1, the largest possible affinity score. In practice, we have not seen the need for more than 1 round. After the seeds selection, we have M clusters, each containing 1 vertex. (2) *Sorted lists construction.* For each seed vertex T_i, we create a descending list of all the vertices that fall into the same affinity graph as T_i. (3) *Cluster enlargement.* This step repetitively iterates through all clusters until all vertices are partitioned. In each iteration, it randomly selects a vertex from the current cluster, and includes the vertex that, among all remaining vertices, has the largest affinity score with this vertex, which can be determined in constant time with the sorted lists produced in step 2.

In our implementation, this graph partitioning happens in parallel on CPU. Its time complexity, in the worst case when all vertices fall into one graph, is $O(N^2 logN)$ (N for number of job chunks). But in practice, as graphs are never very large, the algorithm terminates quickly shown in the next section.

6. EVALUATIONS

We focus our experiments on answering the following main questions:
(1) How much potential does spatial scheduling enabled by SM-centric transformation have?
(2) How much overhead does SM-centric transformation have?
(3) How much benefit can it bring in practice with the simple support outlined in the previous section?

To that end, we implement the two use cases of SM-centric transformation as described in Section 4: One is SM partition for multi-kernel co-runs, and the other is affinity-based scheduling for single-kernel runs. The implementation integrates the solutions described in Section 5. For comparison, we also implement the persistent threads with the best efforts to support these two use cases.

6.1 Methodology

Benchmarks.

Given that the focus of our use cases are on enhancing memory performance, we need a set of memory intensive programs for the validation. Meanwhile, for a comprehensive assessment of the applicability of our techniques, the benchmark set should consist of programs of a broad range of domains, and have a good coverage of both regular and irregular programs. For these reasons, we select nine benchmarks to form our test set. As Table 1 shows, these programs come from four benchmark suites, cover a broad set of domains, and include a similar number of regular and irregular programs. Those irregular benchmarks impose special challenges for GPGPU optimization, and have drawn a lot of attention from the community recently [6, 24, 29, 30, 43, 45].

We give a brief description for these benchmarks. IRREG (a partial differential solver kernel) and NBF (a molecular dynamics kernel) were rewritten to CUDA from C benchmarks [15]. These two benchmarks were studied heavily by previous work [11, 14, 39, 42]. MD and SPMV are both from the SHOC benchmark suite developed by Oak Ridge National Laboratory [10]. CFD from Rodinia benchmark suite [7] simulates fluid dynamics. MM and REDUCE taken from the CUDA SDK samples represent two compute-intensive applications used widely in real-world. We also take NN and PF from the popular Rodinia benchmark suite for a broader coverage.

Co-runs of Kernels.

As current GPUs cannot support the co-existence of two different contexts yet, following prior work [33], we combine two programs into one and use two separate CUDA streams to execute the kernels of the two original programs. Since which kernels run together depends on the practical context, we co-run each pair of the benchmarks for a comprehensive coverage. We use two metrics, System Throughput (STP) and Average Normalized Turnaround Time (ANTT), proposed in [12] and used in [33]. STP shows overall throughput of the whole system, and ANTT shows programs' responsiveness. We measure the execution time of kernel executions and the extra overhead introduced by the transformation (if any) for the calculation of STP and ANTT. Since we are only interested in the overlapped execution, we modify the approach proposed by Tuck and Tullsen [40] and immediately invoke a kernel after it finishes until both kernels are invoked at least 7 times. The last instance of the kernel invocation that finishes later than the other co-run kernel is discarded, because the execution of the last instance of the kernel invocation is not fully overlapped.

Versions based on Persistent Threads.

We compare SM-centric transformation with persistent threads using both SM partition and affinity-based scheduling. As aforementioned, persistent threads by itself cannot directly dictate mappings between jobs and SMs. But with careful designs, it could still support SM partition and affinity-based scheduling, although the support is very limited and requires an awkward implementation. Our specific implementations are as follows.

For SM partition between two co-running kernels, we generate $N_1 (1 \leq N_1 \leq M)$ persistent CTAs for kernel 1, and $(M - N_1)$ persistent CTAs for kernel 2 (where M is the number of SMs in the GPU). In this way, if the hardware scheduler happens to assign one CTA onto each SM, the two kernels would run on different sets of SMs, and the SM partition is materialized. Given that an even distribution is not guaranteed by hardware schedulers, in our experiments, we repeat the experiments many times and use only the results when the distribution happens to be even (performance under uneven distributions is much worse as some SMs are left idle). In order to maintain a good amount of parallelism, each CTA is set to the largest allowable size for the particular kernel, as prior usage of persistent threads often does [9, 44]. Our experiments enumerate all possible partitions (i.e., all values of N_1), and the best performance in these settings is used to compare with the performance of SM-centric transformation results.

We employ a similar idea to let persistent threads support affinity-based scheduling of single-kernel runs. The launch of a kernel creates M (i.e., number of SMs) persistent CTAs. We again use the performance measured only in the runs where the CTAs are evenly distributed on the SMs. When creating the job queue for a persistent CTA, we try to put into the queue the jobs from the same affinity group as identified with the method in Section 5.2. We again make each persistent CTA as large as allowed such that the maximum number of jobs from the CTA queue could get concurrently executed by the CTA. This method, in effect, makes the jobs run concurrently on the same SM, just as what affinity-based scheduling aims to achieve—but only to a limited degree, subject to the number of jobs a CTA can concurrently execute.

Machine Environment.

We run all workloads on an NVIDIA M2075 GPU with CUDA runtime 4.2, compiled by NVCC with the highest optimization level. The host machine has an Intel 8-core Xeon X5672 CPU and 48 GB main memory and runs 64-bit Redhat enterprise 6.2. Without notice, each reported timing result is an average of 10 repeated measurements, and includes all overhead incurred by the transformed code.

6.2 Results in Co-Runs

Figure 3 shows the speedup in terms of ANTT brought by the optimized co-runs respectively supported with SM-centric transformation and persistent threads. The baseline is the traditional and default way to concurrently execute the original kernels. The speedup is defined as the ratio of the optimized ANTT to the original ANTT. "SMC Predicted" and "SMC Optimal" represent the speedups from the SM-centric transformation with, respectively, the parameters predicted by the online model and the best parameters found through offline exhaustive search. We observe the potential speedup because of SM-centric transformation 1.36X. Our prediction model successfully exploits most of the potential by providing 1.33X speedup on average. Three co-runs benefit from the SM-centric transformation substantially with a potential of more than 1.8X speedup. The results validate that SM-centric transformation with the prediction model better exploits the SM and cache resources than the default co-runs. We also observe that the improvement of ANTTs varies across benchmarks. In some cases (e.g., the co-run of mm and reduce), the optimized co-runs have around 2% slowdown. There are two plausible rea-

Table 1: Benchmarks

Benchmark	Source	Description	Irregular
irreg	Maryland [15]	partial diff. solver	Y
nbf	Maryland [15]	force field	Y
md	SHOC [10]	molecular dynamics	Y
spmv	SHOC [10]	sparse matrix vector multi.	Y
cfd	Rodinia [7]	finite volume solver	Y
nn	Rodinia [7]	nearest neighbor	N
pf	Rodinia [7]	dynamic programming	N
mm	CUDA SDK [31]	dense matrix multiplication	N
reduce	CUDASDK [31]	reduction	N

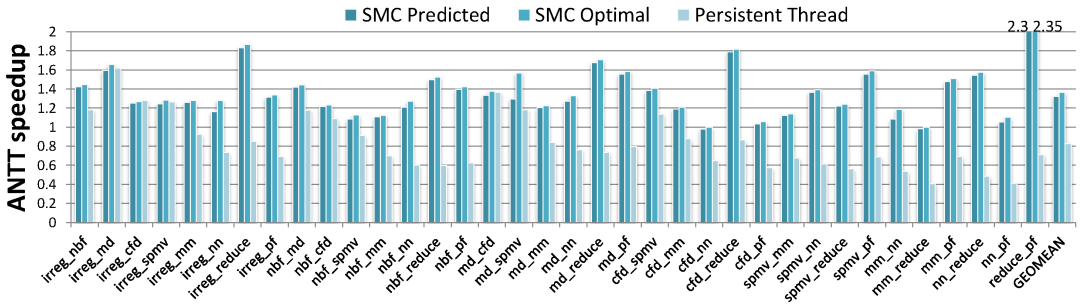

Figure 3: Speedup of average normalized turnaround time.

sons. First, the kernels already have good scalability and hence reducing the number of SMs allocated to them significantly degrades their performance. Second, since those kernels efficiently use shared memory, they do not heavily rely on L1 cache's performance. For example, when MM's thread blocks size is 256, one matrix element is reused 16 times after being loaded into shared memory.

Persistent threads perform much worse than SM-centric transformation. Its best partition leads to more than 50% ANTT degradation for 3 co-run programs. On average, we observe 17% slowdown. The main reason comes from the rigid control of parallelism in persistent threads. As the previous subsection describes, without the capability for a direct control of the job-to-SM mapping, the design of persistent threads support is subject to some restrictions on the number of a CTAs and their size, which cause suboptimal performance on the kernels.

Figure 4 provides the results on system throughput. The baseline is of the same case in Figure 3. SM-centric transformation offers up to 71% (an average of 37%) improvement on STP. The predicted configurations exploit the potential well by providing an average improvement of 33%. We did not observe any throughput degradation for the optimized co-runs, but some co-runs (e.g., cfd and mm) have trivial throughput improvement due to the same reasons as explained for the worsened ANTTs. As known [12], ANTT and STP measure different aspects of a co-run execution; a better ANTT does not always mean a better STP. For instance, the co-run of reduce and pf has the largest ANTT speedup of 2.3X, but its STP improvement is below the average.

Different from the results on ANTT, persistent threads produce an average of 11% STP improvement. The influence of persistent threads on ANTT varies greatly across co-run programs, yielding results between 63% slowdown and 64% improvement. As explained for the increased ANTT, We observe non-trivial throughput loss for some programs because of the suboptimal level of parallelism. Overall, the results showed that SM-centric transformation is a much better choice for SM partitioning than persistent threads.

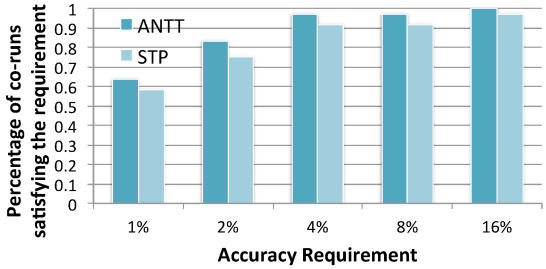

Figure 5: Prediction accuracy.

The SM-centric results also indicate that the simple method for predicting configurations outlined in Section 5.1 is sufficient for SM-centric transformation to effectively support SM partition. To get a direct measure of the method's effectiveness, we report in Figure 5 its accuracy in predicting the suitable configurations. The percentage on the X axis shows the accuracy requirement of the predicted configuration. To be more specific, $P\%$ means that the predicted configuration outperforms at least $N \times (1 - P\%)$ configurations, where $N (N = 36$ in this evaluation) is the total number of configurations. The bar height shows the percentage of co-runs whose predicted configuration satisfies the accuracy requirement. So the bars on the right should be higher than the bars on the left, because a larger percentage on the X axis indicates a more relaxed requirement. For ANTT, when the accuracy requirement is 1%, 63% of co-runs satisfy it. Note that 1% is a harsh requirement, as only the optimal configuration can satisfy it in a limited configuration space. If we relax the requirement to 2%, 83% of co-runs satisfy it, showing a high prediction accuracy. When the requirement

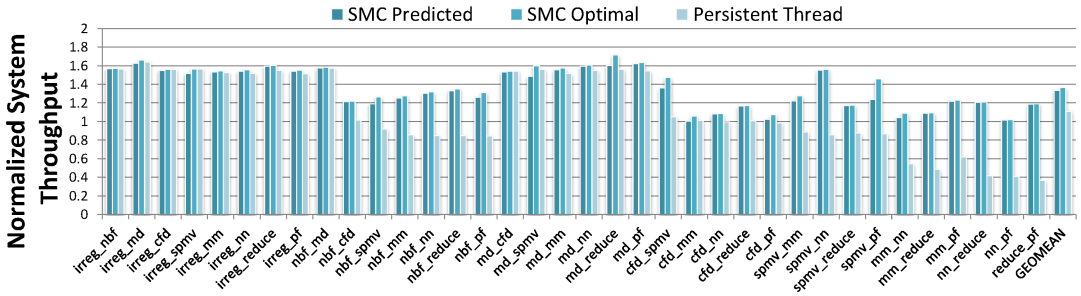

Figure 4: Improvement on system throughput.

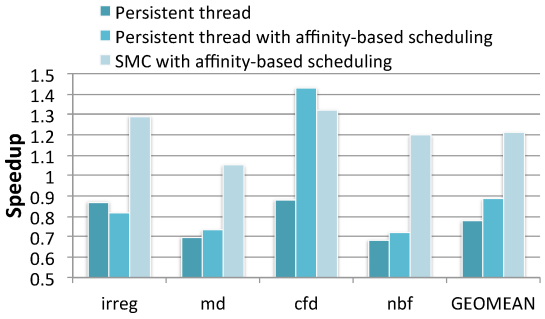

Figure 6: Speedups of single-kernel runs.

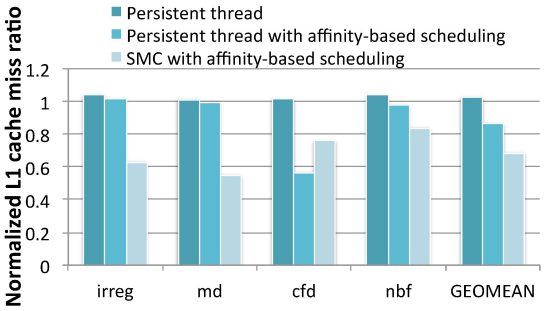

Figure 7: Normalized L1 miss ratios.

is 16%, we notice that the predicted configuration of every co-run satisfies the requirement. For STP, the prediction accuracy is a bit lower, but over 90% of co-runs satisfy the accuracy requirement of 4%. The results echo our improvement on ANTT and STP and show that a simple online model suffices to yield reasonably good configurations.

6.3 Results in Single-Kernel Runs

We also evaluate persistent threads and SM-centric transformations on single-kernel runs. We consider 4 programs (md, irreg, cfd and nbf), as they show a significant level of non-uniform data sharing and rely heavily on the data cache's performance due to their irregular memory access pattern.

Figure 6 provides the speedup results for single-kernel runs of four benchmarks over the original code. Without affinity-based scheduling, persistent threads suffer from insufficient parallelism and produce 22% performance degradation. Affinity-based scheduling improves its performance and reduces the average degradation to 15%. The results indicate that persistent threads, unlike SM-centric transformation, fails to achieve a good balance between parallelism and locality: Keeping one active thread block on each SM enables scheduling jobs with lots of data sharing to one SM, but due to the limitation of the block size, does not have enough concurrent active threads to fully explore the computing power. On the contrary, SM-centric transformation's precise control enables us to find a better trade-off between parallelism and locality, leading to an average of 21% speedup.

Figure 7 shows the L1 cache performance improvement obtained through CUDA hardware performance monitors. The reduction of the cache miss ratios shows the trends largely aligning with the speedup trends. It confirms that L1 cache performance is critical to irregular applications, and the parallelism control and affinity-based scheduling enabled by SM-centric transformation exploit L1 data cache more effectively than the default scheduling does. CFD is an exception, on which, the SM-centric approach performs less well than the persistent threads with affinity-based scheduling. A plausible reason is the effects of warp scheduling, which is out of the control of SM-centric scheduling but could sometimes affect the cache performance substantially.

6.4 Overhead from the SM-centric Transformation

SM-centric transformation adds extra code to the kernels. To quantify the overhead, for each benchmark we run the transformed kernel (with the same number of active threads as the default runs of the original kernels have) but without affinity-based scheduling, whose execution time is denoted as T_{trans}. The overhead is defined as $(T_{trans} - T_{org})/T_{org}$, where T_{org} is the execution time of the original kernel. Figure 8 provides the overhead results. We notice that the overhead can be non-trivial for some benchmarks (e.g., 6.5% for pf) due to two reasons. First, the transformation introduces atomic operations and extra memory accesses to obtain the mapping decision data. Second, the aggregation (i.e., the enhanced version of the transformation) introduces a loop, which does not exist in the original kernels. On average, the overhead from the transformation is 2.8%, but as the previous results show, the overhead is substantially outweighed by the overall benefits.

7. DISCUSSION

Currently, the SM-centric optimization works only on CUDA programs. The reason is that other GPU programming models, such as OpenCL, do not yet support run-time retrieval of

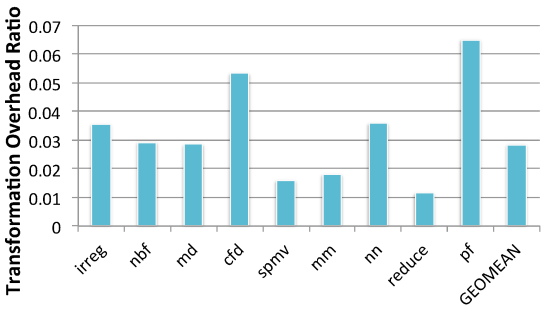

Figure 8: The percentage of overhead from SM-centric transformation.

compute unit identifier. It is our hope that with the evidence provided in this work, OpenCL community will add such a functionality in the near future. Once OpenCL provides a similar interface, the proposed optimization techniques can be easily extended to cover OpenCL programs.

OpenCL has introduced the concept of sub-device, in order to allow partitions of computing units by wrapping a subset of computing unites into a sub-device. It is however supported on only multicore CPUs. Moreover, the partitioning is at the level of OpenCL kernels or applications rather than threads, making it hard to use to manage scheduling of threads or tasks within the same kernel (e.g., the affinity-based scheduling mentioned in Section 4). SM-centric transformation offers a simple, unified solution that supports both scheduling and resource partition, and is ready to use on all GPUs that allows SM ID retrieval.

Although it might be risky to say that future GPUs will never offer interface to allow software control of thread scheduling, there is no evidence showing that such support is coming anytime soon. As a simple pure-software solution that requires no extra hardware cost and complexity, the SM-centric method has its special appeal and practical value.

The job selection component determines the job-to-SM mapping before the kernel is invoked. This fixed mapping could incur load-unbalance due to the non-uniformity in SM processing capability and jobs. It is possible to detect the load-unbalance during the sampling phase. The framework can then invoke the original kernel if load-unbalance happens. Some kernels may change dramatically across invocations in terms of execution time and memory access intensity, creating challenges to sampling-based approaches. Combination with program phase detection and prediction [36, 37] could help address such complexities.

8. RELATED WORK

Prior software methods for task scheduling on GPU mainly concentrate on persistent threads with either static or dynamic (e.g., job stealing) partition of task sets [3, 9, 13, 38, 41, 44]. As aforementioned, although persistent threads and task queues together facilitate task scheduling to a certain degree, they cannot precisely control which tasks run on which SM. It is because tasks are assigned to threads rather than SMs, and the mapping between threads and SMs is determined by hardware (with randomness). As our experiments have shown, for the precise control of task assignment to SMs, the SM-centric transformation offers new opportunities for enhancing data locality and multi-kernel co-run performance, outperforming persistent threads-based optimizations significantly.

There are some studies on changing hardware schedulers for performance, such as the large warp architecture by Narasiman and others [28], the two-level warp scheduler by Jog and his colleagues [18, 19], and the thread block scheduler by Kayiran and others [20]. The SM-centric scheduling is a software solution to the restrictions of hardware schedulers, orthogonal to these hardware approaches. There are some software scheduling works published before, the focus of which have been dealing with the load balance between CPUs and GPUs through task scheduling [4, 26].

Recent years have seen an increasing interest in supporting concurrent executions of GPU kernels. Pai et al. [33] observed significant resource under-utilization during concurrent kernel executions. They proposed elastic kernels to balance resource usage among concurrent kernels through fine-grained resource control. According to the authors of elastic kernels, the technique does not control SM partitions, and cannot be applied to kernels that use shared memory [33]. The spatial scheduling enabled in this work is complementary to elastic kernels, in the sense that it is not subject to the shared-memory limitation, and it improves co-run performance from a different angle, spatial scheduling. These two techniques can be used together. Adriaens and others [2] proposed hardware extensions to partition SMs to different applications for more efficient resource utilization, and evaluated it on a simulator. Zhong and He [48] proposed a runtime system, named Kernelet, which slices kernels into sub-kernels and schedule them for better resource control. It does not enable spatial scheduling of the GPU. CTA aggregation itself is not new. Earlier work has used a similar idea to control resources a kernel uses [33]. It is for the first time used for supporting spatial scheduling.

Many studies have been proposed to improve GPU memory performance, including data placement in memory [8] or on chip [23], streamlining irregular memory accesses or control flows at runtime [43, 45, 46], bypassing L1 cache [17] and so on. The precise control of task-SM affinity enabled by SM-centric transformation opens new opportunities, as illustrated by the affinity-based scheduling.

9. CONCLUSION

This paper presents SM-centric transformation, a simple method that for the first time offers a systematic solution to enable program-level spatial scheduling on the GPU. It reveals the potential of the enabled scheduling control for executions of both single-kernel runs and multi-kernel co-runs. It lists some main challenges for leveraging spatial scheduling on the GPU, and develops a set of practical solutions. It opens up opportunities for leveraging scheduling for optimizing GPU executions.

Acknowledgement This material is based upon work supported by DOE Early Career Award and the National Science Foundation (NSF) under Grant No. 1464216, No. 1320796 and CAREER Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF.

10. REFERENCES

[1] NVIDIA CUDA. `http://www.nvidia.com/cuda`.

[2] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The Case for GPGPU Spatial Multitasking. In *Proceedings of HPCA*, 2012.

[3] T. Aila and S. Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, 2009.

[4] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4), Jan. 2013.

[5] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. Gpuverify: A verifier for gpu kernels. In *OOPSLA*, 2012.

[6] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *IISWC*, 2012.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[8] G. Chen, B. Wu, D. Li, and X. Shen. Porple: An extensible optimizer for portable data placement on gpu. In *Proceedings of MICRO*, 2014.

[9] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic load balancing on single-and multi-gpu systems. In *IPDPS*, 2010.

[10] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU*, 2010.

[11] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI*, 1999.

[12] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3), May 2008.

[13] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing*, 2012.

[14] H. Han and C. W. Tseng. Improving locality for adaptive irregular scientific codes. In *LCPC*, 2000.

[15] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *TPDS*, 17(7):606–618, 2006.

[16] S. Herbert and D. Marculescu. Characterizing chip-multiprocessor variability-tolerance. In *DAC*, 2008.

[17] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and improving the use of demand-fetched caches in gpus. In *ICS*, 2012.

[18] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated scheduling and prefetching for gpgpus. In *ISCA*, 2013.

[19] A. Jog, O. Kayiran, N. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU performance. In *ASPLOS*, 2013.

[20] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT*, 2013.

[21] J. Lee, V. Sathisha, M. J. Schulte, K. Compton, and N. S. Kim. Improving throughput of power-constrained gpus using dynamic voltage/frequency and core scaling. In *PACT*, 2011.

[22] S. Lee, T. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *LCPC*, pages 539–553, 2003.

[23] C. Li, Y. Yang, Z. Lin, and H. Zhou. Automatic data placement into gpu on-chip memory resources. In *Proceedings of CGO*, 2015.

[24] J. Li, G. Tan, M. Chen, and N. Sun. Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *PLDI*, 2013.

[25] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu programs optimization. In *IPDPS*, pages 1–10, 2009.

[26] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, 2009.

[27] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *PACT*, 1999.

[28] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *MICRO*, 2011.

[29] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus topology-driven irregular computations on gpus. In *IPDPS*, 2013.

[30] R. Nasre, M. Burtscher, and K. Pingali. Morph algorithms on gpus. In *PPOPP*, 2013.

[31] NVIDIA. Cuda software development toolkit v4.2. https://developer.nvidia.com/cuda-toolkit-42-archive.

[32] NVIDIA. Nvidia's next generation cuda computer architecture: Fermi.

[33] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *ASPLOS*, 2013.

[34] NVIDIA Parallel Thread Execution. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[35] X. Shen, Y. Gao, C. Ding, and R. Archambault. Lightweight reference affinity analysis. In *ICS*, Cambridge, MA, June 2005.

[36] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 165–176, 2004.

[37] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of International Symposium on Computer Architecture*, pages 336–349, San Diego, CA, June 2003.

[38] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the gpu. *ACM Transactions on Computer Systems*, 33(6), 2014.

[39] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *PLDI*, 2003.

[40] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading pentium 4 processor. In *PACT*, 2003.

[41] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the gpu. In *Proceedings of the Conference on High Performance Graphics*, 2010.

[42] B. Wu, E. Zhang, and X. Shen. Enhancing data locality for dynamic simulations through asynchronous data transformations and adaptive control. In *PACT*, 2011.

[43] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *PPoPP*, 2013.

[44] S. Xiao and W. chun Feng. Inter-block gpu communication via fast barrier synchronization. In *IPDPS*, 2010.

[45] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. In *ASPLOS*, 2011.

[46] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining gpu applications on the fly. In *ICS*, pages 115–125, 2010.

[47] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. Grace: A low-overhead mechanism for detecting data races in gpu programs. In *PPoPP*, 2011.

[48] J. Zhong and B. He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *CoRR*, abs/1303.5164, 2013.

[49] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI*, 2004.