Reducing Contention in Shared Last-Level Cache for Throughput Processors

HSIEN-KAI KUO and BO-CHENG CHARLES LAI, National Chiao-Tung University
JING-YANG JOU, National Central University and National Chiao-Tung University

Deploying the Shared Last-Level Cache (SLLC) is an effective way to alleviate the memory bottleneck in modern throughput processors, such as GPGPUs. A commonly used scheduling policy of throughput processors is to render the maximum possible thread-level parallelism. However, this greedy policy usually causes serious cache contention on the SLLC and significantly degrades the system performance. It is therefore a critical performance factor that the thread scheduling of a throughput processor performs a careful trade-off between the thread-level parallelism and cache contention. This article characterizes and analyzes the performance impact of cache contention in the SLLC of throughput processors. Based on the analyses and findings of cache contention and its performance pitfalls, this article formally formulates the aggregate working-set-size-constrained thread scheduling problem that constrains the aggregate working-set size on concurrent threads. With a proof to be NP-hard, this article has integrated a series of algorithms to minimize the cache contention and enhance the overall system performance on GPGPUs. The simulation results on NVIDIA's Fermi architecture have shown that the proposed thread scheduling scheme achieves up to 61.6% execution time enhancement over a widely used thread clustering scheme. When compared to the state-of-the-art technique that exploits the data reuse of applications, the improvement on execution time can reach 47.4%. Notably, the execution time improvement of the proposed thread scheduling scheme is only 2.6% from an exhaustive searching scheme.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memory, Shared memory*; D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Throughput processors, thread-level parallelism, cache contention, shared last-level cache, thread scheduling

ACM Reference Format:
Hsien-Kai Kuo, Bo-Cheng Charles Lai, and Jing-Yang Jou. 2014. Reducing contention in shared last-level cache for throughput processors. ACM Trans. Des. Autom. Electron. Syst. 20, 1, Article 12 (November 2014), 28 pages.
DOI: http://dx.doi.org/10.1145/2676550

1. INTRODUCTION

The recently emerging throughput-oriented architectures dedicate most of the chip area to thousands of simpler and smaller computing cores. Such architectures have been proved to achieve great performance by leveraging the numerous computing cores and the massive Thread-Level Parallelism (TLP) of applications. Up to the present, there already have been plenty of such processors proposed by academic research

This work was supported in part by the National Science Council, Taiwan, under the grant NSC 102-2220-E-009.
Authors' addresses: H.-K. Kuo (corresponding author) and B.-C. C. Lai, National Chiao Tung University, 300, Hsinchu City, East District, Taiwan; email: hsienkai.kuo@gmail.com; J.-Y. Jou, National Central University, Taiwan and National Chiao Tung University, 300, Hsinchu City, East District, Taiwan.

DOI: http://dx.doi.org/10.1145/2676550

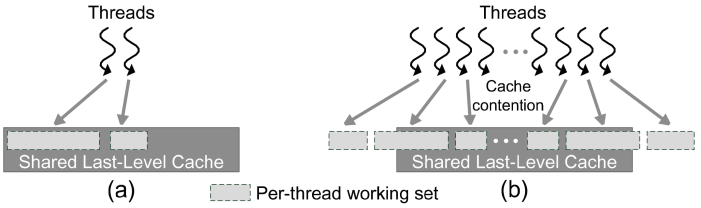

Fig. 1. (a) Low-thread-level parallelism execution without cache contention; (b) massive thread-level-parallelism execution with cache contention.

as well as commercial product vendors, such as Rigel [Johnson et al. 2011], Sun's Sparc T3 [Shin et al. 2010], and NVIDIA's Fermi and Kepler GPGPU [Nvidia 2012a; Wittenbrink et al. 2011]. Even with the numerous computing cores and massive TLP, the peak performance of throughput processors is usually limited by the long latency of off-chip memory accesses. To alleviate the memory bottleneck, a Shared Last-Level Cache (SLLC) has been adopted in the latest throughput-oriented architectures to better utilize the data locality inherent in applications [Nvidia 2012a; Wittenbrink et al. 2011]. By keeping the frequently reused data on chip while allowing different computing cores to share or exchange data, the SLLC can potentially mitigate the performance penalty caused by the long latencies of off-chip memory accesses.

Nowadays, NVIDIA's GPGPU has been become a widely used throughput processor that implements deep TLP with SLLC to achieve sheer computation throughput [Garland and Kirk 2010; Keckler et al. 2011; Nickolls and Dally 2010]. In such throughput-oriented architectures, allowing a high degree of TLP could boost the computation throughput from massive concurrent execution; however, the concurrent execution of a large amount of threads might also result in serious *cache contention* on the SLLC. The contention on the SLLC could cause a large number of cache misses that would generate a lot of off-chip memory traffic to the DRAM subsystem. This issue could significantly degrade the system throughput. According to the measurements in Section 3, without considering the cache contention on the SLLC, the number of DRAM accesses can be increased by 11.8x.

In an SLLC, cache contention happens when its cache capacity is insufficient to support all the concurrent threads. In this situation, threads start replacing the useful data of each other. As cache contention occurs, it generates a tremendous number of unnecessary cache misses in an SLLC. These misses are converted into a lot of long-latency off-chip memory accesses and therefore cause considerable execution time and energy consumption. Moreover, the cache contention issue can be aggravated when a throughput processor exposes a high degree of TLP and executes tens of thousands of concurrent threads, such as NVIDIA's Fermi and Kepler GPGPU [Nvidia 2012a; Wittenbrink et al. 2011]. Figures 1(a) and 1(b) illustrate examples of how the cache contention impairs the memory system performance by introducing numerous cache misses. As shown in Figure 1(a), the SLLC capacity is large enough for low-TLP execution. However, as shown in Figure 1(b), when massive TLP are executed at the SLLC, the cache capacity may be insufficient to support all the threads. These threads therefore start competing with each other for the insufficient cache resource. Consequently, numerous undesired capacity misses and conflict misses are generated. As a result, not only could the potential computation throughput of TLP be seriously compromised, but also the energy consumption could increase significantly.

This article has demonstrated that avoiding cache contention on the SLLC is a critical performance factor and should be considered as one of the main optimization goals. According to the measurements shown in Section 3, without considering the cache contention on the SLLC, the performance of a throughput processor can be degraded

by 2.3x and the energy consumption can be increased by 3.2x. Nevertheless, the cache-contention-related optimization has not been extensively studied for throughput processors. Most of the existing studies focus on optimizing the memory access behavior by exploiting specific memory features, such as utilizing scratch-pad memory [Deng et al. 2009], enhancing memory coalescing by thread clustering [Kuo et al. 2012], or data and computation reordering [Wu et al. 2013; Zhang et al. 2011a]. With these approaches, cache contention can be somehow mitigated while the memory access behavior is optimized. However, without an appropriate characterization and optimization, the cache contention can still happen and impair the system performance even after applying these techniques. This is especially true when the sheer amount of concurrent threads share a relatively small SLLC on throughput processors. Consequently, leveraging the massive parallelism of TLP on enormous processing cores while appropriately avoiding cache contention in an SLLC has become a critical design concern to achieve superior performance and energy efficiency on a throughput processor.

This article concentrates on studying the cache contention avoidance of an SLLC in throughput processors. To our best knowledge, this article is the first study that characterizes the cache contention of an SLLC in throughput processors. Based on the characterization, this article is also the first attempt to directly minimize the cache contention of an SLLC by considering the *aggregate working-set size*. The term "aggregate working-set size" refers to the amount of demanded memory space of a set of threads that need to be serviced by an SLLC. In summary, this article has the following main contributions. First, this article performs comprehensive characterization on the cache contention issue of the SLLC for throughput processors. Based on the characterization, this work conducts the performance analysis of cache contention and provides the fundamentals for further optimizations. Second, based on the analysis, this research formulates a generic thread scheduling problem for avoiding the cache contention of the SLLC in throughput processors, called the *aggregate working-set-size-constrained thread scheduling problem*. This article also proves that this problem is NP-hard. Third, to handle different architectural considerations, this article analyzes several variants of the generic problem and proposes the thread scheduling algorithms for these variants. Note that one of the variants implements the current architecture of NVIDIA's Fermi GPGPUs.

The simulation results on NVIDIA's Fermi architecture have shown that the proposed thread scheduling scheme achieves an average of 46.8% reduction in cache misses and 31.8% execution time enhancement over a commonly used thread clustering scheme. For applications with more threads and higher complexity, the execution time improvement can reach up to 61.6%. When compared to the state-of-the-art scheme that exploits the data reuse inherent in applications, the average improvement on execution time is 22.9% and up to 47.4% in applications with a more complex workload. Notably, the execution time improvement of the proposed thread scheduling scheme is only 2.6% from an exhaustive searching scheme.

The rest of the article is organized as follows. Section 2 introduces the background of this work. Section 3 characterizes the cache contention issue in an SLLC, while Section 4 introduces the problems, challenges, and potential of the aggregate working-set-size-constrained thread scheduling. The details of *aggregate working-set-size-constrained thread scheduling* are discussed in Section 5. Section 6 shows the implementation details and experimental results. Section 7 surveys the related work and, finally, Section 8 concludes this article.

2. BACKGROUND

This section introduces the background of this article. Section 2.1 introduces the throughput-oriented architecture and the application characteristics are discussed in Section 2.2.

2.1. Throughput-Oriented Architecture

This section introduces a representative throughput-oriented architecture, namely NVIDIA Fermi [Wittenbrink et al. 2011] and its programming environment CUDA [NVIDIA 2012]. The Fermi architecture implements several per-Stream Multiprocessor (SM) L1 caches of size up to 48KB, and a unified 768KB L2 cache. Each L1 cache is shared among all the cores in an SM. The unified L2 cache is the Shared Last-Level Cache (SLLC) shared by all the SMs in a GPGPU. In the CUDA environment, a parallel application is described as a series of *kernels*. Each kernel contains a large number of independent threads that are assembled into multiple thread groups, called Cooperative Thread Arrays (CTAs), which can be concurrently executed. The kernels are then executed sequentially based on the invocation order. In this way, the data dependency can be maintained between kernels.

During the execution of a kernel, CTAs are dynamically dispatched to SMs by the gigathread scheduler [Wittenbrink et al. 2011]. In order to achieve the maximum TLP and hide the long latency of off-chip memory accesses, the gigathread scheduler dispatches as many CTAs as possible to each SM, and performs context switching when threads encounter stalls caused by off-chip memory accesses. With this scheme, a great number of threads are executed in a time-multiplexing manner. Hence, the system resources, such as registers, processing cores, and load/store (LD/ST) units in an SM, can be extensively utilized with less idling. Note that, in the Fermi architecture, the number of CTAs dispatched to an SM can be configured for different launches of a kernel, but is fixed during the execution of a kernel.

Recently, several thread schedulers with dynamic reconfigurable TLP have been proposed [Kayiran et al. 2012; Rogers et al. 2012]. An architecture with such schedulers has the ability to change the number of concurrent CTAs during the kernel execution. This article models this extension and integrates it into the GPGPU-Sim simulator [Bakhoda et al. 2009]. To reflect the behavior of a real architecture synchronization is also considered when reconfiguring the number of concurrent CTAs during the kernel execution. The synchronization operation would first suspend the current execution in a GPGPU architecture and then resume the execution after changing the number of concurrent CTAs. A certain amount of execution cycles are needed for a synchronization operation.

2.2. Application Characteristics on Throughput Processors

The applications running on throughput processors in general inherently have massive computation parallelism. However, the actual performance of these applications is vastly determined by the associated memory access behavior. To facilitate the following discussion on the memory access behavior of applications on throughput processors, this article categorizes the applications into two types, namely regular and irregular, according to their memory access behavior. In regular applications, the cache contention optimization is relatively straightforward because of the regularly coordinated memory accesses. For example, the Fast Fourier Transform (FFT) and matrix multiplications generate regular memory access behavior [NVIDIA 2012]. For this type of regular applications, programmers can easily analyze and predict the concurrent execution as well as the memory access behavior. However, such optimization becomes considerably more challenging when dealing with applications with uncoordinated memory accesses. In contrast with regular applications, irregular applications usually contain uncoordinated memory accesses and the sharing of data is nonuniformly behaved across different threads. Meanwhile, in the irregular application, the working sets of threads also tend to be varying and nonuniformly distributed. This article aims at tackling the challenges of cache contention minimization in irregular

applications, although, without losing generality, the proposed approach can also be applied to regular applications. This article collects a variety of irregular applications from different fields, namely Electronic Design Automation (EDA) [Kuo et al. 2012], Molecular Dynamics (MD) [Han and Tseng 2006], and Computational Fluid Dynamics (CFD) [Das et al. 1994]. Because of the irregularity, these applications often need an input file to declare the required data in each thread. For example, the input netlist file is needed in the gate-level logic simulation [Kuo et al. 2012]. Depending on different application characteristics, the proposed scheduling approach can be performed before or during the execution of an application. In the situation that the working sets of threads behave statically for a given input file, this article can be used as an offline optimization framework. For each application, the scheduling process only needs to be performed once for the static execution characteristics of kernel calls. The overhead of the scheduling can be amortized by the numerous following iterations of kernel calls. Alternatively, the proposed scheduling can also be used as an online optimization framework. The scheduling process is periodically performed to adapt to the changing of the working sets and execution behavior throughout all the iterations of kernel calls. In this case, one needs to consider the overhead of the online scheduling process, and it would require enough iterations of kernel calls to attain performance enhancement.

3. CHARACTERIZING CACHE CONTNETION

To address the cache contention issue, this research proposes an analysis method that can characterize cache contention in a quantitative manner. The analysis lays out a solid foundation to characterize the cache contention of a throughput processor, and provides guidelines for further research and optimization. Section 3.1 introduces an approach for measuring and quantifying cache contention. Section 3.2 conducts the measurement and analysis of the impact of cache contention on different performance metrics, while Section 3.3 further summarizes the analyses and findings.

3.1. Quantifying Cache Contention

To understand the performance implications of cache contention, this section introduces a quantitative measurement to reflect the level of cache contention. This approach will then be used for the measurements and analysis in Section 3.3. In order to deploy such quantitative measurement, Definition 1 defines the *aggregate working-set size* that captures the amount of cache space that needs to be provided by an SLLC for a set of threads.

Definition 1 (*Aggregate Working-Set Size*). Given a set of threads and corresponding working-set sizes, the *aggregate working-set size* of such set of threads is defined as the accumulation of the working-set sizes.

In Definition 1, the *Aggregate Working-Set Size* (AWSS) of a set of threads is defined as the accumulated working-set size. That is to say, the working sets of threads are assumed disjointed, or only overlapped slightly. This assumption will be demonstrated as valid from the measurements and analysis in Section 3.2. Based on Definition 1, Definition 2 introduces the *Cache Contention Ratio* (CCR) that provides an indication of the severity of the cache contention for a given set of threads.

Definition 2 (*Cache Contention Ratio*). Given a set of threads and the aggregate working-set size, the *cache contention ratio* of such a set of threads is defined as the ratio of the aggregate working-set size to the capacity of shared cache.

$$CCR = \frac{AWSS}{Cache\_cap}$$

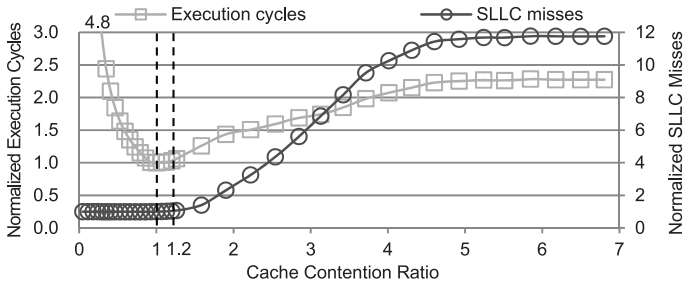

Fig. 2. The comparison of execution cycles and SLLC misses with respect to cache contention ratio based on the euler application. Numbers are normalized to the measurement that cache contention ratio equals one.

CCR: Cache contention ratio
$AWSS$: Aggregate-working-set-size of a set of threads
$Cache\_cap$: Capacity of shared cache

In Definition 2, the cache contention ratio is defined as the ratio of the aggregate working-set size to the capacity of shared cache. As defined in Definition 1, the aggregate working-set size refers to the amount of demanded memory space of a set of threads that needs to be serviced by an SLLC. With Definition 2, one can assess the level of cache contention for a given set of threads. When the value of the cache contention ratio is far higher than one, a serious cache contention is expected because the cache capacity is deficient for the set of threads. Otherwise, when the value of the cache contention ratio is below one, it means the set of threads receives sufficient cache space for holding their data and therefore the cache contention is expected to be very slight.

3.2. Measurement and Analysis

To understand the performance implications of cache contention, this section performs measurement on several performance metrics, including SLLC misses, execution cycles, and energy consumption. In the measurement of each performance metric, the number of concurrent threads is adjusted to achieve different aggregate working-set size and cache contention ratios. Here the cache contention ratio is obtained by the ratio of the aggregate working-set size to the capacity of an SLLC. In Figures 2 and 3, the measurement is conducted by using the *euler* application, which will be one of the benchmarks used in Section 6. As shown in Figure 2, when the cache contention ratio is below one, the capacity of the SLLC is enough to support all the concurrent threads. Hence, one can find very small numbers of SLLC misses in this region. In the region where the ratio is within 20% larger than one, the capacity of the SLLC tends to match the requirement of concurrent threads. Therefore, the SLLC misses still tend to remain at the same level with just very slight increase. However, when the cache contention ratio is further raised, cache contention starts taking place because the capacity of the SLLC is insufficient for all the concurrent threads. As a result, the number of SLLC misses drastically increases and reaches a saturated level when the ratio is within five to six. According to Figure 2, like the SLLC misses, the execution cycles reflect very similar behavior when the cache contention ratio is larger than one. The only difference happens when the cache contention ratio is below one. In this region, the number of concurrent threads is still too low to leverage the TLP, although the SLLC capacity is enough for the concurrent threads. Consequently, the number of execution cycles gets larger as the cache contention ratio decreases.

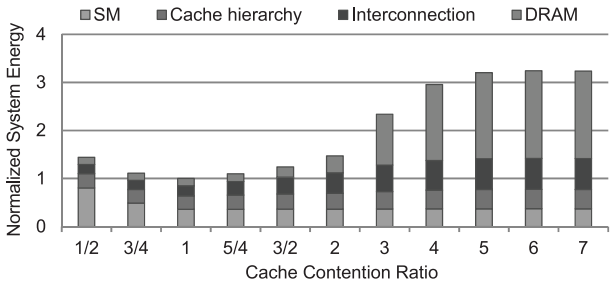

Fig. 3. The comparison of energy consumption breakdown with respect to cache contention ratio based on the euler application. Numbers are normalized to the measurement that cache contention ratio equals one.

To further understand the impact of cache contention on the system energy consumption, Figure 3 illustrates the energy consumption breakdown for different cache contention ratios. In Figure 3, due to the lower number of execution cycles and SLLC misses, one can find that the lowest energy consumption is achieved when the cache contention ratio is around one. Conversely, when the ratio decreases, the SM energy consumption becomes higher due to the longer execution time, while on the other side, when the cache contention ratio increases, the large amount of SLLC misses causes significant data request traffic on the interconnection network which consumes more energy. Moreover, the SLLC misses also generate lots of accesses to DRAM. As a result, the energy consumption of DRAM is drastically increased.

3.3. Summary of Analyses and Findings

By referring to the cache contention ratio in Definition 1 and the measurement and analysis discussed in Section 3.2, one can find that the aggregate working-set size of concurrent threads is highly correlated to the cache contention level in an SLLC. Meanwhile, it also has significant impact on both computation throughput and energy consumption of throughput processors. To be more specific, scheduling threads without the awareness of its aggregate working-set size could cause severe cache contention that generates up to 11.8x more misses in an SLLC and hurts the execution time and energy consumption up to 2.3x and 3.3x, respectively. According to these findings, the aggregate working-set size has become an important consideration for SLLC performance optimization. The key optimization goal is to achieve superior performance by rendering the maximum possible TLP while avoiding cache contention in an SLLC. To address such an optimization requirement, this article introduces a scheduling approach for minimizing cache contention. By controlling the execution behavior of concurrent threads, the proposed approach can effectively constrain the aggregate working-set size of the threads and thus minimize cache contention in an SLLC. This scheduling approach called *aggregate working-set-size-constrained thread scheduling* and will be formally formulated in Section 5.

4. EXAMPLES OF AGGREGATE WORKING-SET-SIZE-CONSTRAINED THREAD SCHEDULING

This section uses a series of simple examples to illustrate the main concept behind this approach. As the example in Figure 4 shows, an application contains a set of CTAs A to L with different working-set sizes. Assume the architecture allows at most four concurrent CTAs and the capacity of the SLLC is ten. Recall that the aggregate working-set size of a set of CTAs is defined as the accumulated working-set size. For example, the aggregate working-set size for the set of CTAs {C, E, G} is six. As shown in Figure 4(a), in order to maximize the TLP, a straightforward scheduling scheme is to assign the maximum number of CTAs at each scheduling step. This simple greedy

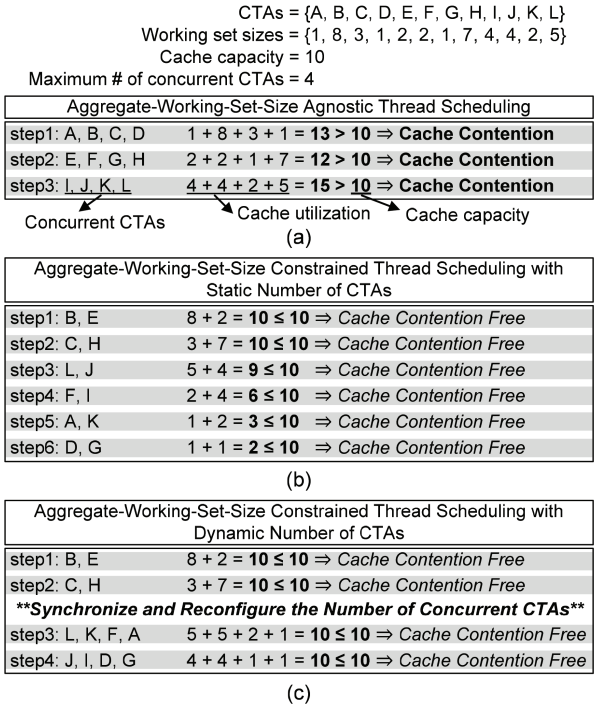

Fig. 4. (a) The example of aggregate working-set-size-agnostic thread scheduling; (b) the example of aggregate working-set-size-constrained thread scheduling with the architectural limitation; (c) the example of aggregate working-set-size-constrained thread scheduling with flexibility to adjust the number of concurrent CTAs.

thread scheduling scheme is commonly used in current throughput processors, such as NVIDIA's Fermi and Kepler architectures [Nvidia 2012a; Wittenbrink et al. 2011]. As discussed in Section 3, the cache contention level of a set of concurrent CTAs is correlated to its aggregate working-set size. Without the awareness of this circumstance, a scheduling scheme could result in a schedule of CTAs where the aggregate working-set size is far larger than the available capacity in the SLLC. As shown in Figure 4(a), at the schedule from step 1 to step 3, severe cache contention is induced during the execution of these scheduling steps. The cache contention will be aggravated when a throughput processor schedules a large number of concurrent CTAs with higher aggregate working-set size. Besides, the diverse working-set sizes in applications could make the cache contention an even more intricate problem.

To avoid cache contention, the aggregate working-set size needs to be constrained. An aggregate working-set-size-constrained scheme schedules CTAs to steps in which the aggregate working-set size is constrained to be no larger than the cache capacity. As mentioned in Section 2.1, the current throughput processors only allow a fixed number of concurrent CTAs during kernel execution. A static number of concurrent CTAs should be decided during thread scheduling. Figure 4(b) shows an example of this case. In this example, the number of concurrent CTAs is reduced to a static number and the corresponding scheduling result can successfully mitigate the cache contention. Nevertheless, when the number of concurrent CTAs is fixed throughout a kernel, one cannot assign more CTAs to a step, even though the SLLC still has the capacity to accommodate more CTAs. For example, step 4 in Figure 4(b) still has four more

available cache space units. Also, fixing the number of concurrent CTAs could yield more scheduling steps that might result in longer overall execution time.

To address the preceding two concerns, one may expect the architecture to enable the flexibility to adjust the number of concurrent CTAs during kernel execution. This article refers to this flexibility as the reconfiguration of the number of concurrent CTAs. Figure 4(c) depicts an example of such a scenario. In this example, the aggregate working-set-size-constrained scheme reconfigures the number of concurrent CTAs to be four after step 2. Starting from step 3, more CTAs are allowed to be scheduled. Therefore, steps 3 and 4 can contain four CTAs. However, as mentioned in Section 2.2, such a reconfiguration incurs a synchronization operation between scheduling steps. The synchronization operation would first suspend the current execution in a throughput processor and then resume the execution after changing the number of concurrent CTAs. As a matter of course, performing a synchronization operation consumes a certain amount of execution cycles determined by the implementation. Hence, in order to achieve the best schedule of CTAs, the aggregate working-set-size-constrained scheme should also take the cost of reconfiguration into account. In summary, the prior discussions on the application- and architecture-related features indicate that the *aggregate working-set-size-constrained thread scheduling* is a nontrivial and critical approach to achieve superior performance in throughput processors.

5. AGGREGATE WORKING-SET-SIZE-CONSTRAINED THREAD SCHEDULING

This section introduces aggregate working-set-size-constrained thread scheduling, including the problem formulation and proposed algorithms. Section 5.1 formulates this problem according to the throughput-oriented architectures. Section 5.2 proposes an algorithm to solve the problem with a general formulation. Section 5.3 illustrates this problem on the architecture of a Fermi GPGPU system that can be solved as a special case of the general formulation. Section 5.4 further extends the discussion to appropriately relax the constraint for thread scheduling, while Section 5.5 integrates all the variants of thread scheduling algorithms and proposes a thread scheduling framework to manage different thread scheduling constraints and architectural parameters.

5.1. Problem Formulation

To formulate the aggregate working-set-size-constrained thread scheduling problem, let $c^n = \{c_1, c_2, \ldots, c_n\}$ denote a given collection of CTAs. Each CTA has a working-set size $w : c^n \rightarrow \mathbb{R}_+$. For the sake of simplicity, CTAs are sorted so that $w(c_1) \leq w(c_2) \leq \cdots \leq w(c_n)$. The thread scheduling mechanism assigns CTAs into a schedule $s^m = \{s_1, s, \ldots, s_m\}$. A schedule contains several scheduling steps from s_1 to s_m. Each scheduling step can contain one or more CTAs and the CTAs in the same scheduling step will be concurrently dispatched to SMs. In this article, the *Thread-Level Parallelism* (TLP) corresponds to the total number of CTAs that are concurrently executing on SMs. Therefore, the TLP will be adjusted at the granularity of CTAs. Eq. (1) lists the corresponding TLP of the SMs when executing a scheduling step s_i.

$$TLP(s_i) = |\{c_j | c_j \in s_i\}| \tag{1}$$

Whereas, as in Definition 1, the Aggregate Working-Set Size (AWSS) refers to the amount of demanded cache space of the SLLC for a set of CTAs, the aggregate working-set size of a scheduling step s_i is listed as in Eq. (2).

$$AWSS(s_i) = \sum_{c_j \in S_i} w(c_j) \tag{2}$$

According the characterization and analysis in Section 3, the aggregate working-set size is highly correlated to the cache contention level. In order to avoid cache contention,

the aggregate working-set size of a scheduling step is constrained so as not to exceed the SLLC capacity. This performance consideration is known as the *aggregate working-set-size-constraint* and listed in Definition 3. In Definition 3, this constraint is satisfied if the aggregate working-set size of a scheduling step is smaller than the SLLC capacity.

Definition 3 (*Aggregate Working-Set-Size Constraint*). Given a scheduling step s_i containing a set of CTAs, the *aggregate working-set-size constraint* is satisfied if the aggregate working-set size of the scheduling step is smaller than the SLLC capacity.

Under the aggregate working-set-size constraint, the objective of the scheduling is to find a schedule s^m while the total execution time is minimized. In a given schedule s^m, there are two causes deciding the span of its execution time. The first is the total number of scheduling steps m. According to the modern Single-Instruction Multiple-Threads (SIMT) execution model of GPGPU architectures, threads are deeply time-multiplexed to execute the same sequence of instructions but on different data. In this article, CTAs are assumed to contain threads among which the data reuse and divergence are appropriately optimized. Hence, the behavior of a CTA and the corresponding execution time turn out to be very similar to other CTAs. Since the CTAs have very similar execution time, the aggregated execution time is proportional to the total number of scheduling steps m. Therefore, more scheduling steps will take longer execution time. The second cause is from the cost of changing the TLP between steps. According to Section 2.2, a synchronization operation is required when changing the TLP from one scheduling step to the next. Eq. (3) shows that the synchronization operation is needed only when the concurrency of a scheduling step s_i is different from the TLP of its next step s_{i+1}. In other words, no synchronization is required when the two consecutive steps s_i and s_{i+1} apply the same TLP. Note that Eq. (3) returns a number 1 when a synchronization operation is require.

$$Sync\,(s_i, s_{i+1}) = \begin{cases} 1, & TLP\,(s_i) \neq TLP\,(s_{i+1}) \\ 0, & else \end{cases} \tag{3}$$

For a given schedule s^m, one can obtain the total number of synchronization operations $tot\_Sync(s^m)$ by simply traversing the schedule as listed in Eq. (4).

$$tot\_Sync\,(s^m) = \sum_{i=1}^{m-1} Sync\,(s_i, s_{i+1}) \tag{4}$$

In a throughput processor, a synchronization operation impairs the execution time in various ways [Feng and Xiao 2010]. In this article the term *Cost Per Synchronization* (*CPS*) is used to express the overall cost of a single synchronization operation. It is a performance factor that combines the impact of two sources, namely the number of scheduling steps and the synchronization cost. In this article, CPS is normalized to the average execution time of CTAs, and is therefore a number between zero and a certain constant CPS_{max} as shown in Eq. (5).

$$CPS \in \mathbb{R} : 0 < CPS \leq CPS_{max} \tag{5}$$

Combining Eqs. (4) and (5), one can evaluate the overall cost of synchronization through Eq. (6).

$$Sync\_cost(s^m) = CPS \times tot\_Sync(s^m) \tag{6}$$

With the preceding discussion, the aggregate working-set-size-constrained thread scheduling problem is then formulated as in Problem 1.

Problem (*Aggregate Working-Set-Size-Constrained Thread Scheduling Problem*). Given a collection of CTAs $c^n = \{c_1, c_2, \ldots, c_n\}$ with working-set size $w : c^n \rightarrow \mathbb{R}_+$

and SLLC capacity *Cap_SLLC*, the *aggregate working-set-size-constrained thread scheduling problem* is to find a schedule $s^m = \{s_1, s, \ldots, s_m\}$ where every scheduling step has an aggregate working-set size smaller than *Cap_SLLC*, such that the number of scheduling steps and the synchronization cost are minimized.

$$
\begin{aligned}
&\textbf{\textit{Minimize}} && \textit{the number of scheduling steps and total synchronization cost} \\
&&& = m + Sync\_cost\,(s^m) \\
&\textbf{\textit{Subject to}} && \forall s_i : TLP\,(s_i) \leq TLP_{max} \\
&&& \forall s_i : AWSS\,(s_i) \leq Cap\_SLLC \\
&&& \forall s_i \neq s_j : s_i \cap s_j = \emptyset \\
&&& s_1 \cup s_2 \cup \cdots \cup s_m = c^n
\end{aligned}
$$

As in Problem 1, the aggregate working-set-size-constrained thread scheduling problem is formulated to minimize the number of scheduling steps of s^m and the total cost of synchronization subject to: (1) the TLP in each scheduling step cannot exceed the maximum possible TLP, namely TLP_{max} and meanwhile the aggregate working-set size in each scheduling step should be within the SLLC capacity, namely *Cap_SLLC*; and (2) each CTA is assigned to exactly one scheduling step. The following discussion gives an example of this thread scheduling problem on NVIDIA's Fermi architecture. As mentioned, in a kernel, the cache contention issue in an SLLC heavily depends on its aggregate working-set size. This issue is modeled by the first constraint in Problem 1. Recall that, in the current GPGPU architecture, the TLP is fixed during the execution of a kernel. To further optimize TLP, splitting the execution of the original kernel into several kernels is one of the commonly used methods to vary the TLP during runtime. Each of the split kernels completes a portion of the tasks but with different TLP. Under the constraint on aggregate working-set size, it is possible to perform kernel splits to increase the TLP for higher computation throughput. However, the splits of a kernel would suffer the cost of launching extra kernels. These two issues are modeled by the first and second objective factors in Problem 1. According to the previous discussion, the aggregate working-set-size-constrained thread scheduling problem is therefore to assign CTAs into a series of kernels to minimize the total execution time, including both the execution time and launch cost of kernels, while satisfying the aggregate working-set-size constraint.

From Lemma 1 that follows and its proof, one can conclude that such a scheduling problem is NP-hard. It is therefore very difficult, if not impossible, to obtain an optimal solution in polynomial time.

LEMMA 1. *Given a collection of CTAs* $c^n = \{c_1, c_2, \ldots, c_n\}$ *with working-set size* $w : c^n \rightarrow \mathbb{R}_+$ *and SLLC capacity Cap_SLLC, it is NP-hard to find a schedule* s^m *such that the number of scheduling steps and the total cost of synchronization are minimized while every scheduling step has an aggregate working-set size smaller than Cap_SLLC.*

PROOF. A well-known NP-hard problem, the *bin packing problem* [Garey et al. 1972] can be reduced to the aggregate working-set-size-constrained thread scheduling problem. The bin packing problem is formulated as the following: Let $a_1, a_2, \ldots, a_n$ be a given collection of items with sizes $s\,(a_i) > 0, 1 \leq i \leq n$. A bin packing problem divides all the items into a minimum number of blocks, called *bins*, subject to the sum of the sizes of items in each bin being at most a given capacity. The reduction can be derived because the bin packing problem is basically a special case of the aggregate working-set-size-constrained thread scheduling problem, where synchronization takes no cost. □

5.2. Thread Scheduling Algorithm

In Problem 1, the general aggregate working-set-size-constrained thread scheduling problem, the thread scheduling algorithm has the flexibility to change the TLP between scheduling steps. With this flexibility, the scheduling algorithm can pack more CTAs into one step to minimize the number of scheduling steps. However, the rearrangement of TLP also incurs synchronization where the corresponding cost should also be minimized. Lemmas 2 and 3 show the observations that become useful properties to develop a thread scheduling algorithm.

LEMMA 2. *For any schedule s^m to the aggregate working-set-size-constrained thread scheduling problem, the execution time of the schedule s^m, $m+Sync\_cost(s^m)$ is less than or equal to $m \times (CPS_{max} + 1) - CPS_{max}$.*

PROOF. The detailed proof is derived in Eq. (7). First, the number of synchronization operations is bounded by $m-1$. Next, by applying the constraint of CPS in Eq. (5), one can find that $m + Sync\_cost(s^m)$ is less than or equal to $m \times (CPS_{max} + 1) - CPS_{max}$. □

$$Sync\_cost\,(s^m) \leq CPS \times (m-1)$$

$$\Rightarrow Sync\_cost\,(s^m) \leq CPS_{max} \times (m-1)\,, when\ 0 < CPS \leq CPS_{max}$$

$$\Rightarrow m + Sync\_cost\,(s^m) \leq m \times (CPS_{max} + 1) - CPS_{max}, when\ 0 < CPS \leq CPS_{max} \quad (7)$$

In Lemma 2, for any schedule s^m, the overall execution time, including the number of scheduling steps m and the cost of synchronization $Sync\_cost(s^m)$, is at most $m \times (CPS_{max} + 1) - CPS_{max}$. Since the CPS_{max} is a constant (as will be discussed in Section 6.2), this property implies that the minimum number of steps m presents a theoretical bound on the execution time of a schedule s^m. Therefore, minimizing the number of steps m becomes a good optimization goal to the problem.

Even though Lemma 2 delivers a useful property for designing the thread scheduling algorithm, it still cannot directly minimize the cost of synchronization. In order to minimize the cost of synchronization, this article shows another interesting finding that derives and guarantees the minimum number of synchronizations in a given schedule s^m.

LEMMA 3. *Given a schedule s^m in which the scheduling steps are permutable, the number of synchronization operations of s^m, namely $tot\_Sync(s^m)$, is minimum if the scheduling steps are sorted by the TLP.*

PROOF. According to Eqs. (3) and (4), considering a schedule s^m with a certain order of scheduling steps, the number of synchronization operations equals the number of transitions of TLP between adjacent steps. Therefore, sorting yields the order with the minimum number of transitions and thus the minimum number of synchronization operations. □

For a given schedule s^m, Lemma 3 shows that sorting s^m by the TLP of each scheduling step yields the minimum number of the synchronization operations and thus the minimum cost of synchronization. Also, one can find that the sorting process does not affect the number of scheduling steps m.

According to Lemma 2, the thread scheduling algorithm is designed to first minimize the number of scheduling steps m without the consideration of synchronization. Next, Lemma 3 is leveraged to minimize the number of synchronization operations. As a result, by means of Lemma 2, the synchronization in Problem 1 can be removed and the problem turns out to be very similar to a variant of the classical bin packing problem, called *k-cardinality bin packing problem* [Krause et al. 1975]. In the k-cardinality

ALGORITHM 1: Aggregate Working-Set-Size-Constrained Thread Scheduling

1: $k \leftarrow TLP_{max}$
2: $cap \leftarrow Cap\_SLLC$
3: $s^m \leftarrow k\text{-}Cardinality\_Bin\_Packing\ (c^n, cap, k)$
4: *sort* s^m *by TLP to minimize* $Sync\_cost(s^m)$
5: $s^{m'} \leftarrow s^m$
6: **while** $cost(s^{m'}) \leq cost(s^m)$ **do**
7: $s^m \leftarrow s^{m'}$
8: $k \leftarrow k - 1$
9: $s^{m'} \leftarrow k\text{-}Cardinality\_Bin\_Packing\ (c^n, cap, k)$
10: *sort* $s^{m'}$ *by TLP to minimize* $Sync\_cost(s^{m'})$
11: **end while**
12: **return** s^m

bin packing problem, the number of *items*, called *cardinality*, that can be placed in a bin is at most k. In the past few decades, the bin packing problem and its variants have been extensively studied. Although it is difficult to get the best solution due to its NP-hard nature, its acceptable solutions can still be found in polynomial time by using approximation or heuristic methods. Several existing polynomial-time algorithms have been proven with low approximation ratio. This research leverages two well-studied algorithms, *Largest Memory First* (LMF) and *Iterated Worst-Case Decreasing* (IWFD) [Krause et al. 1975], to solve the aggregate working-set-size-constrained thread scheduling problem. Note that one can also utilize other k-cardinality bin packing algorithms to solve this problem.

The thread scheduling algorithm is shown in Algorithm 1. In Algorithm 1, the k is first set as the maximum possible TLP of a scheduling step. Lines 3 to 5 obtain an initial solution and lines 6 to 10 iteratively try different k to see if any possible improvement can be achieved. In each iteration, by using k-cardinality bin packing algorithms, the number of scheduling steps m is first minimized without the consideration of synchronization. After that, the schedule s^m is sorted to minimize the number of synchronization operations. The algorithm terminates when a local optimum has been reached.

5.3. Handling Static Thread-Level Parallelism

In order to deal with different architectural limitations, this section further discusses the thread scheduling problem for handling the static TLP. Recall that, in NVIDIA's Fermi architecture, the number of allowed CTAs of each SM is static throughout the execution of a kernel. This architectural limitation is referred to as the *static thread-level parallelism constraint* and listed in Definition 4. In Definition 4, the aforesaid is satisfied if all the scheduling steps in an arbitrary schedule s^m have the same and static TLP. This is to say, a thread scheduling algorithm has no way to change the TLP between scheduling steps.

Definition 4 (*Static Thread-Level Parallelism Constraint*). Given a schedule s^m, the *static thread-level parallelism constraint* is satisfied if every scheduling step has the same TLP.

This finding implies that there requires no synchronization in the schedule s^m. This is because, according to Eq. (3), a synchronization operation is needed only when adjusting the TLP between two steps. With the *static thread-level parallelism constraint* as described in Definition 4, the term $Sync\_cost(s^m)$ in Problem 1 can be removed to achieve the *static TLP and aggregate working-set-size-constrained thread scheduling problem* as in Problem 2.

Problem 2 (*Static TLP and Aggregate-Working-Set-Size-Constrained Thread Scheduling Problem*). Given a collection of CTAs $c^n = \{c_1, c_2, \ldots, c_n\}$ with working-set size $w : c^n \rightarrow \mathbb{R}_+$ and SLLC capacity *Cap_SLLC*, the *static TLP and aggregate working-set-size-constrained thread scheduling problem* is to find a schedule $s^m = \{s_1, s, \ldots, s_m\}$ where every scheduling step has the same TLP and an aggregate working-set size smaller than *Cap_SLLC*, such that the number of scheduling steps is minimized.

$$
\begin{aligned}
\textbf{\textit{Minimize}} \quad & \textit{the number of scheduling steps} = m \\
\textbf{\textit{Subject to}} \quad & \forall s_i : TLP(s_i) \leq TLP_{max} \\
& \forall s_i \neq s_j : TLP(s_i) = TLP(s_j) \\
& \forall s_i : AWSS(s_i) \leq Cap\_SLLC \\
& \forall s_i \neq s_j : s_i \cap s_j = \emptyset \\
& s_1 \cup s_2 \cdots s_m = c^n
\end{aligned}
$$

Problem 2, the static TLP and aggregate working-set-size-constrained thread scheduling problem, is formulated to minimize the number of scheduling steps of s^m subject to the following: (1) every scheduling step has the same TLP that cannot exceed the maximum possible TLP, namely TLP_{max}; (2) the aggregate working-set size in each scheduling step cannot exceed the SLLC capacity, namely *Cap_SLLC*; and (3) each CTA is assigned to exactly one scheduling step.

With the formulation of Problem 2, one can find that it is essentially like the k-cardinality bin packing problem [Krause et al. 1975]. Hence, one can leverage any existing algorithms to solve the static TLP and aggregate working-set-size-constrained thread scheduling problem. The thread scheduling algorithm is shown in Algorithm 2. In line 1 of Algorithm 2, the TLP k is first set to the maximum number of concurrent CTAs of the architecture. Lines 2 to 7 decide the appropriate TLP by finding the largest acceptable k without violating the aggregate working-set-size-constraint. Once the appropriate k is decided, any k-cardinality bin packing algorithm can be applied to solve this problem.

ALGORITHM 2: Static TLP and Aggregate Working-Set-Size-Constrained Thread Scheduling

1: $k \leftarrow TLP_{max}$
2: **repeat**
3: $\quad cap \leftarrow w(c_n) + w(c_{n-1}), \ldots, w(c_{n-k+1})$
4: $\quad k \leftarrow k - 1$
5: **until** $Cap\_SLLC \geq cap$
6: $cap \leftarrow Cap\_SLLC$
7: $s^m \leftarrow k\text{-}Cardinality\_Bin\_Packing(c^n, cap, k)$
8: **return** s^m

5.4. Relaxing Aggregate Working-Set Size Constraint

With the aggregate working-set size constraint as detailed in Definition 3, a thread scheduling algorithm avoids arranging too high a TLP that might cause serious cache contention. Nevertheless, in order to satisfy the hard constraint on aggregate working-set size, it can be too restrictive on the TLP to efficiently make use of the last piece of cache space in every scheduling step. To address this disadvantage, the aggregate working-set size constraint could be appropriately relaxed. In the meantime, the relaxation should also be careful to keep the nature of the aggregate working-set size constraint. The measurement in Section 3 provides an interesting observation that

becomes a useful property to develop such a relaxation. According to the discussions in Section 3, for a scheduling step of a set of CTAs, the cache contention is very minor when the corresponding cache contention ratio is slightly larger than one. In other words, a slight violation of the aggregate working-set-size-constraint would not induce stiff penalty from cache contention. With this property, the aggregate working-set-size-constraint is further relaxed as in Definition 5.

Definition 5 (*Relaxed Aggregate Working-Set-Size-Constraint*). Given a scheduling step s_i containing a set of CTAs, the relaxed aggregate working-set size constraint is satisfied if there exists a designated CTA in the scheduling step such that the removal of this CTA brings the aggregate working-set size of the scheduling step within the SLLC capacity.

In Definition 5, different from the ordinary aggregate working-set-size-constraint, the relaxed aggregate working-set-size Constraint is still satisfied if the removal of a designated CTA brings the aggregate working-set size within the SLLC capacity. Under the relaxed aggregate working-set size constraint, a thread scheduling algorithm has the flexibility to fully make use of the cache capacity in every scheduling step. As a result, the performance is improved by increasing the TLP while avoiding severe cache contention. With the relaxed aggregate working-set size constraint in Definition 3, Problem 1 can be rewritten the relaxed aggregate working-set size-constrained thread scheduling problem as outlined in Problem 3.

In Problem 3, the relaxed aggregate working-set-size-constrained thread scheduling problem is formulated to minimize the number of scheduling steps of s^m and the total cost of synchronization subject to the following: (1) the TLP in each scheduling step cannot exceed the maximum possible TLP, namely TLP_{max}; (2) the aggregate working-set size after the removal of a designated CTA in each scheduling step cannot exceed the SLLC capacity, namely Cap_SLLC; and (3) each CTA is assigned to exactly one scheduling step.

Problem 3 (*Relaxed Aggregate Working-Set-Size-Constrained Thread Scheduling Problem*). Given a collection of CTAs $c^n = \{c_1, c_2, \ldots, c_n\}$ with working-set size $w : c^n \rightarrow \mathbb{R}_+$ and cache capacity of SLLC Cap_SLLC, the *relaxed aggregate working-set-size-constrained thread scheduling problem* is to find a schedule $s^m = \{s_1, s, \ldots, s_m\}$ where the removal of a designated CTA in every scheduling step brings its aggregate working-set size smaller than Cap_SLLC, such that the number of scheduling steps and the synchronization cost are minimized.

$$
\begin{aligned}
\textit{Minimize} \quad & \textit{the number of scheduling steps and total synchronization cost} \\
& = m + Sync\_cost\,(s^m) \\
\textit{Subject to} \quad & \forall s_i : TLP\,(s_i) \leq TLP_{max} \\
& \forall s_i : \exists c_j \in s_i \,|AWSS(s_i - c_j) \leq Cap\_SLLC \\
& \forall s_i \neq s_j : s_i \cap s_j = \emptyset \\
& s_1 \cup s_2 \cup \cdots \cup s_m = c^n
\end{aligned}
$$

Similar to Algorithm 1, Lemmas 2 and 3 are also leveraged to design the thread scheduling, in which the number of scheduling steps m is first minimized without the consideration of synchronization and then the number of synchronization operations is minimized. Different from Algorithm 1, in order to handle the relaxed aggregate working-set-size-constraint, another variant of the bin packing problem, namely the open-end bin packing problem, is considered. In the open-end bin packing problem, a bin can be filled to a level exceeding its capacity so long as there is a designated

last item in the bin such that the removal of this item brings the bin's level back to below its capacity. There exist several polynomial-time approximation algorithms proposed for this problem. These algorithms include *Mixed Fit* (MXF) and *Greedy LookAhead Next Fit* (GLANF) [Yang and Leung 2003]. This article leverages these existing and well-studied algorithms to solve the relaxed aggregate working-set-size constrained thread scheduling problem. Note that one can also utilize other open-end bin packing algorithms to solve this problem. The same algorithm (i.e., Algorithm 1) can be leveraged for this problem. In order solve it, the open-end bin packing algorithm is used in lines 3 and 9 of Algorithm 1.

ALGORITHM 3: Integrated Thread Scheduling Framework

input: A collection of CTAs $c^n = \{c_1, c_2, \ldots, c_n\}$ and, working set sizes of CTAs $w(c_1), w(c_2), \ldots, w(c_n)$, the capacity of SLLC *Cap_SLLC* and the maximum possible thread-level-parallelism TLP_{max}.

Parameter: 1) Boolean value: *Static_TLP*, for enabling the *Static Thread-Level-Parallelism Constraint*, 2) Boolean value: *Relax_AWSS*, for enabling the *Relaxed Aggregate-Working-Set-Size Constraint*.

Output: A CTA schedule s^m.

1: $k \leftarrow TLP_{max}$
2: **if** *Static_TLP* **then**
3: **repeat**
4: $cap \leftarrow w(c_n) + w(c_{n-1}), \ldots, w(c_{n-k+1})$
5: $k \leftarrow k - 1$
6: **until** $Cap\_SLLC \geq cap$
7: **end if**
8: $cap \leftarrow Cap\_SLLC$
9: **if** *Relax_AWSS* **then**
10: $s^m \leftarrow$ *Open-End_Bin_Packing*(c^n, cap, k)
11: **else**
12: $s^m \leftarrow$ *k-Cardinality_Bin_Packing*(c^n, cap, k)
13: **end if**
14: *sort* s^m *by TLP to minimize Sync_cost*(s^m)
15: $s^{m'} \leftarrow s^m$
16: **while** $cost\ (s^{m'}) \leq cost(s^m)$ **do**
17: $s^m \leftarrow s^{m'}$
18: $k \leftarrow k - 1$
19: **if** $Relax\_AWSS = \text{true}$ **then**
20: $s^{m'} \leftarrow$ *Open-End_Bin_Packing*(c^n, cap, k)
21: **else**
22: $s^{m'} \leftarrow$ *k-Cardinality_Bin_Packing*(c^n, cap, k)
23: **end if**
24: *sort* $s^{m'}$ *by TLP to minimize Sync_cost*$(s^{m'})$
25: **end while**
26: **return** s^m

5.5. Integrated Thread Scheduling Framework

Combining the thread scheduling variants introduced in Sections 5.2 to 5.4, this section proposes an integrated thread scheduling framework in Algorithm 3. In order to deal with the requirements of different types of aggregate working-set-size-constrained thread scheduling, the thread scheduling framework is designed to have the flexibility to: (1) handle the static thread-level parallelism constraint one can explicitly specify the constraint of TLP for the thread scheduling where; and (2) adapt to the relax

aggregate working-set-size-constraint one can explicitly specify whether the relaxation of aggregate working-set size is enabled for the thread scheduling.

In Algorithm 3, with user-specified parameter *Static_TLP*, lines 2 to 7 decide an appropriate setting for TLP. Once the TLP is decided, an initial thread schedule can be obtained as follows. First, according to the condition of the parameter *Relax_AWSS*, lines 9 to 13 select the proper bin packing algorithm to minimize the number of scheduling steps m. Then, the sorting is applied to further minimize the number of synchronization operations in line 14. After this, lines 16 to 25 iteratively try different TLP to see whether any possible improvement can be achieved. In each iteration, by using the appropriate bin packing algorithms, the aggregated execution time of scheduling steps is first minimized without considering the synchronization. The schedule s^m is then sorted to minimize the cost of synchronization operations. The algorithm will terminate when it reaches a local optimum.

6. EXPERIMENTAL EVALUATION

This section discusses the experiment evaluations. Sections 6.1 and 6.2 introduce the implementation details and experiment setup. Further evaluations and analyses are discussed in Sections 6.3 to 6.8.

6.1. Implementation of Thread Scheduling

Figure 5 illustrates the implementation flow of the proposed thread scheduling. Recall that all the threads in a kernel are independent and can be executed concurrently by any available SMs. However, data dependency may still exist between kernels. In order to avoid violating the inter-kernel data dependency, this article takes only one kernel at a time and performs the compilation and thread scheduling on the independent threads in this kernel. Before launching the kernel, threads are first grouped to obtain the collection of CTAs, *numCTA*s, and *threadPerCTA* as in step 1 of Figure 5. Also, the kernel launch in step 1 triggers a process to dynamically compile the kernel code by using gpuocelot, a dynamic compilation framework for GPGPUs [Diamos et al. 2010]. As depicted in step 2, the gpuocelot framework first performs the working-set analysis on the given kernel to obtain the working-set size of each CTA. With the information of working-set sizes, step 3 issues the thread scheduling processes as discussed in Section 5 to generate the CTA schedule s^m. Next, the gpuocelot framework continues the generation of the executable codes for the given kernel as in step 4. Finally, the kernel is launched with the thread scheduling information as shown in step 5. In the practical implementation, in order to enforce a specific CTA schedule s^m, one can leverage the drivers and/or runtime APIs to pass the scheduling information to the GPGPU, or one can also apply *persistent thread* techniques as a pure software-based approach [Gupta et al. 2012]. To simplify the implementation, this article passes the scheduling information through an explicitly specified file to the GPGPU-Sim simulator, a cycle-accurate performance simulator for GPGPU [Bakhoda et al. 2009]. As depicted in Figure 5, when the flag *update* is not set, the proposed flow is statically performed once for the numerous kernel calls. In contrast, when the flag update is set, the flow is applied in a dynamic manner as in step 6. The flow is periodically performed to adapt to the change of working set sizes throughout all the iterations of kernel calls. In this research, the thread scheduling is performed statically because the working-set size is fixed throughout all the iterations of kernel calls. As a result, the working-set analysis only needs to be performed once. The overhead of analysis and thread scheduling can be amortized by the numerous following iterations of kernel calls. For example, a study in Lai et al. [2014] has shown that the Electronic Design Automation (EDA) applications require several thousands of iterations to even break the overhead of a similar prerun

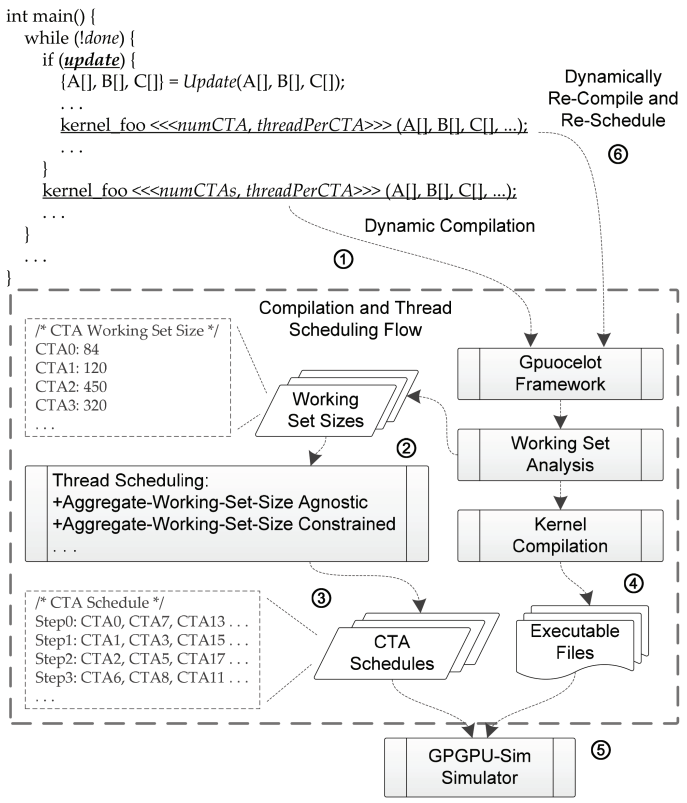

Fig. 5. The example of the implementation flow of the proposed thread scheduling approaches.

analysis. However, the typical number of iterations for the practical usage of EDA applications can easily reach millions.

6.2. Experiment Setup and Benchmarks

The experimental environment is based on Nvidia's Fermi architecture [Wittenbrink et al. 2011] and its CUDA programming environment [NVIDIA 2012b]. Without loss of generality, the proposed scheme can also be applied to the Kepler architectures [Nvidia 2012a] and the OpenCL programming environment [Khronos 2011]. In the baseline architecture, a Fermi architecture is modeled and integrated into the GPGPU-Sim simulator [Bakhoda et al. 2009]. Table I lists the configurations of the baseline architecture. In order to perform the thread scheduling, the scheduling algorithms take the cache capacity parameter, *Cap_SLLC*, which is 768KB as specified in Table I. The cost of the NVIDIA's GPGPU synchronization function *syncthreads*() has been profiled and reported to take about 70 cycles per call as shown in Table I [Wong et al. 2010]. Besides, the resultant energy consumption of different thread scheduling schemes is also evaluated by using the GPUWattch, an architecture-level power model that has been integrated into GPGPU-Sim [Wu et al. 2013]. Table II lists the parameter configuration of GPUWattch.

The proposed thread scheduling schemes are evaluated with a set of irregular applications. In contrast with regular applications, the working sets of threads are varying and nonuniformly distributed. Because of the irregularity, these applications often need input files to declare the required data in each thread. For example, the input netlist

Table I. Baseline Architectural Configurations in GPGPU-Sim

Number of SMs	15
SM configuration	1.4 GHz, 32-wide pipeline, 32 threads per warp, 32768 registers per SM, 1536 threads per SM,Greedy-Then-Oldest (GTO) warp scheduler, **CTAs per SM (reconfigurable TLP, default 8)**
Synchronization	**70 cycles per synchronization**
L1 cache	48 KB per SM, 6 way, 128 byte per line
L2 cache (shared last-level cache)	unified 768 KB, 16 way, 128 byte per line
DRAM	1.84 GHz, 6 GDDR5 channels, 2 chips per channel, 16 banks, 16 entries per chip, FR-FCFS policy
Interconnection network	1.4 GHz, crossbar (15 SMs and 6 memory controllers), 32-byte flit it size

Table II. Baseline Parameter Configurations in GPUWattch

Technology node	40 nm
Temperature	380 K
Device type	ITRS's high performance
Long channel transistor	yes
Interconnect wire model	aggressive

Table III. Irregular and Massively Parallel Applications

Applications	Fields	Descriptions	Data set sizes	Number of kernels
bfs	Electronic Design Automation	Breadth first search	2.6 MB	2
sta		Static timing analysis	3.0 MB	2
gsim		Gate level logic simulation	3.5 MB	2
nbf	Molecular Dynamics	kernel abstracted from the GROMOS	6.3 MB	2
moldyn		Non-bonded force calculation in the CHARMM	10.2 MB	2
irreg	Computational Fluid Dynamics	Kernel of iterative partial differential equation solver	6.3 MB	2
euler		Finite-difference approximations on Eulerian mesh	8.5 MB	1
unstructured		Fluid dynamics with unstructured mesh model	10.2 MB	2

file is needed in the gate-level logic simulation. This article collects a variety of irregular applications from different fields and uses these applications to demonstrate the effectiveness of the proposed scheduling methods. Three EDA applications are adopted from Kuo et al. [2012], and five scientific simulation benchmarks are taken from the COSMIC project [Han and Tseng 2006] and Chaos group [Das et al. 1994]. Table III lists all the applications used in this article. In Table III, the circuit file *b18* (114k gates and 219k interconnections) from ITC'99 circuit suite [Rogers et al. 2012] is applied for the three EDA applications, while the mesh file *foil* (144k nodes and 1074k edges) from the COSMIC project [Han and Tseng 2006] is used for the remaining scientific applications.

To better clarify the discussion, the following terms are used to represent different schemes. The first scheme, referred to as *nvcc+Coal*, applies a simple coalescing optimization technique and uses the NVIDIA's *nvcc* compiler with -O3 optimization level [NVIDIA 2012; Zhang et al. 2011a]. The *nvcc* compiler performs a series of loop and memory optimizations, such as loop strength reduction, loop unrolling, memory space optimization, and rematerialization. Although such a scheme accomplishes coalescing optimization and some transformations within the scope of a single thread, it is unaware of cache contention between threads. The term cluster represents the

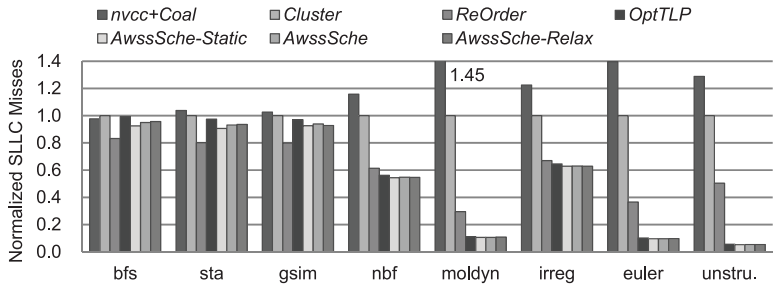

Fig. 6. The SLLC misses comparison of various schemes. Numbers are normalized to the cluster scheme.

essential approach for mitigating cache contention, in which threads are grouped to minimize the working-set size [Kuo et al. 2012]. Based on the cluster scheme, this article evaluates several schemes with different further considerations and optimizations. First, by combining the data and computation reordering techniques [Wu et al. 2013; Zhang et al. 2011a] and data reuse optimization techniques [Chen et al. 2007; Zhang et al. 2011b], this work implements a sophisticated scheme, namely *reOrder*, that adjusts the scheduling order to exploit data reuse between thread groups. The proposed aggregate working-set-size-constrained thread scheduling scheme is represented as *AwssSche*, which takes the aggregate working-set size as a constraint during the thread scheduling. The variants of the thread scheduling are represented as suffixes to *AwssSche*. The static TLP scheduling is abbreviated as *AwssSche-Static*. The relaxed aggregate working-set size scheduling is represent as *AwssSche-Relax*. Finally, *OptTLP* achieves the optimal execution time through exhaustively searching every possible TLP [Kayiran et al. 2012; Rogers et al. 2012].

6.3. Cache Misses, Thread-Level Parallelism and Synchronization Operation

This section discusses the cache misses, thread-level-parallelism, and synchronization operations of different schemes. Figure 6 compares the cache miss reduction of the SLLC, unified L2 cache. Although the *nvcc+Coal* scheme leverages several optimizations in *nvcc* and also optimizes the memory coalescing, it is still unaware of the cache contention of an application. In such a situation, this greatest number of causes very severe cache contention and thus the cache misses among all schemes. By grouping the threads and reducing the corresponding working-set size, the cluster scheme mitigates the cache contention issue and reduces an average 19.5% of the cache misses when compared to the *nvcc+Coal* scheme. Particularly, the cluster scheme reduces more cache misses in applications with heavy memory traffic, such as *moldyn* and *euler* applications. Based on a cluster scheme, the reOrder scheme further adjusts the execution order of threads to exploit the data reuse inherent in applications and reduces an average 39 % of cache misses.

Figure 7 depicts the corresponding TLP of different schemes. Note that the minimum and maximum TLP are also shown for the *AwssSche* and *AwssSche-Relax* schemes. One can find that all of the *nvcc+Coal*, cluster and reorder schemes render the maximum possible TLP to maximize system throughput. However, in such a high-TLP configuration, the cache contention can happen and impairs the system performance, for example, in *moldyn* and *euler* applications. By taking the aggregate working-set size constraint, the *AwssSche* scheme adjusts the TLP during the execution and reduces the cache misses by 46.8% on average. As shown in Figure 6, the *AwssSche* scheme delivers a cache-miss reduction very close to that of the *OptTLP* scheme. Note that the *OptTLP* scheme only considers the optimal execution time, hence can deliver

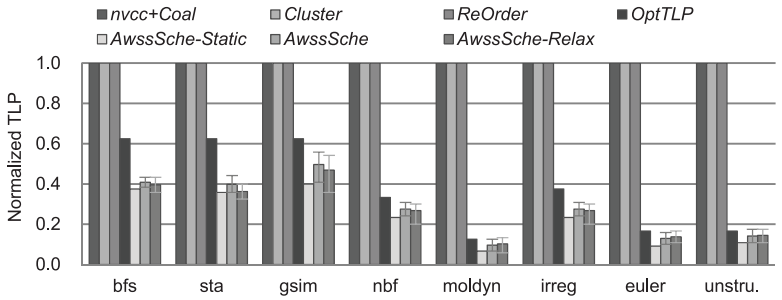

Fig. 7. The TLP comparison of various schemes. Numbers are normalized to the cluster scheme.

Table IV. Comparison of Number of Synchronization Operations and the Cost Per Synchronization

Characteristics		*bfs*	*sta*	*gsim*	*nbf*	*moldyn*	*irreg*	*euler*	*unstructured*
Number of Synchronizations	*AwssSche*	2	3	4	8	7	8	7	8
	AwssSche-Relax	1	1	3	6	7	6	7	7
Cost Per Synchronization (CPS)		0.93	0.82	0.79	0.11	0.01	0.12	0.02	0.01

for slightly more cache misses than other schemes. According to Figure 7, one can find that all the schemes perform limited reduction on the three EDA applications except for the reorder scheme. This is because the three EDA applications have reorder smaller sizes of working sets and only cause very mild cache contention. In this situation, due to the further data reuse consideration, the reorder scheme has higher potential to reduce cache misses. In contrast, the other scientific applications have much larger working sets and also a higher possibility to cause severe cache contention. This gives the *AwssSche* scheme more room to reduce the cache misses.

When considering the static thread-level parallelism constraint, the *AwssSche-Static* scheme can select static TLP and perform the thread scheduling. As shown in Figure 7, when compared to the *AwssSche* scheme, the *AwssSche-Static* scheme selects a lower TLP to satisfy such a static TLP constraint. By contrast, with the relaxed aggregate working-set size constraint, the *AwssSche-Relax* scheme can schedule more CTAs and efficiently make use of the cache capacity of SLLC. Meanwhile, the *AwssSche-Relax* scheme also performs the trade-off between TLP and the number of synchronization operations. Table IV lists the corresponding numbers of synchronization operations of *AwssSche* and *AwssSche-Relax* schemes together with the Cost Per Synchronization (CPS). Because of the relaxed aggregate working-set size constraint, *AwssSche-Relax* has the flexibility to search a larger solution space to achieve higher TLP and fewer synchronization operations. Note that, in some applications, *AwssSche-Relax* selects lower TLP to mitigate the cost of synchronization, such as in the *sta* application. Although these variants have different considerations of TLP and the number of synchronization operations, the cache contention can be successfully alleviated through constraining the aggregate working-set size. The difference in cache miss reduction between variants is within a small range from -1.1% to 2.6%. In a brief summary, we demonstrate an average of 46.8% reduction of cache misses by using the proposed aggregate working-set-size-constrained thread scheduling scheme. In fact, in some applications with heavy memory traffic, the reduction is up to 94.6%.

6.4. Thread Scheduling Overhead, Execution Time and System Energy

This section discusses the overhead of thread scheduling, execution time, and the corresponding system energy. Recall that the overhead can be amortized by the numerous following iterations of kernel calls. Figure 8 compares the overhead of different

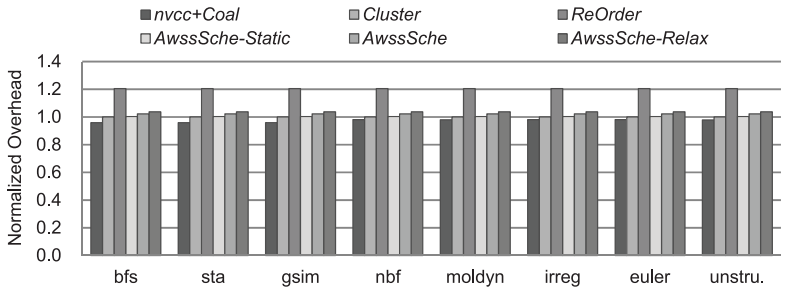

Fig. 8. The overhead comparison of various schemes. Numbers are normalized to the cluster scheme.

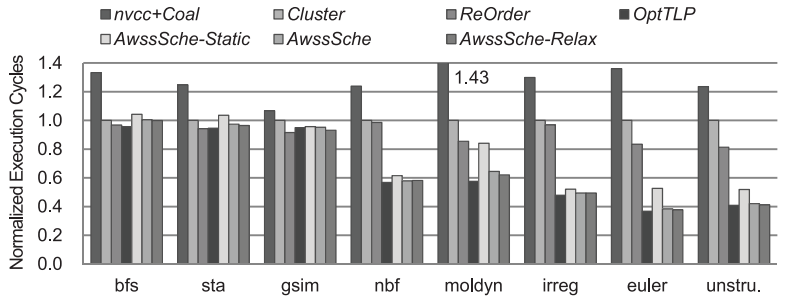

Fig. 9. The execution-cycle comparison of various schemes. Numbers are normalized to the cluster scheme.

schemes. In Figure 8, one can find that all of the schemes have similar overhead except for the reorder scheme. This is because the reorder scheme relies on an analysis of data reuse and performs a data-reuse-oriented scheduling that has higher complexity than the other schemes. This would require more iterations of kernel calls to amortize the extra overhead. On the other hand, *AwssSche* and its variants only cost an average 2.1% of runtime overhead that can be easily amortized with a few iterations of kernel calls. In our implementation, the overheads of *AwssSche* and its variants are under 0.13 seconds. To simplify the discussions and provide a single baseline for further comparisons, a reasonable and yet large enough number of iterations of kernel calls are used for the evaluations in the following paragraphs and sections.

The execution cycles of different schemes are shown in Figure 9, where a strong correlation between cache misses and execution cycles is shown. In general, in the three EDA applications with mild cache contention, all the cache-contention-aware schemes, namely cluster, reorder, and *AwssSche*, have very similar improvement. The reorder scheme achieves a slightly better improvement than the *AwssSche* scheme by further exploiting the data reuse. However, it becomes more complicated in the other scientific applications in which cache contention takes place and impacts the performance significantly. This gives the *AwssSche* scheme more room to improve the execution cycles. Compared to the *nvcc-Coal* scheme, the cluster scheme improves the execution cycle by 27.6% on average, the reorder scheme achieves an average 9% execution-cycle improvement over the cluster scheme. By means of the aggregate working-set size constraint, *AwssSche* delivers an average of 31.8% improvement over the cluster, which is only 2.6% from the *OptTLP* scheme. Compared to *AwssSche*, the *AwssSche-Static* has to render lower TLP and thus achieves less improvement. In contrast, the *AwssSche-Relax* scheme delivers further improvement over the *AwssSche* scheme in some applications, such as *moldyn* and *euler*.

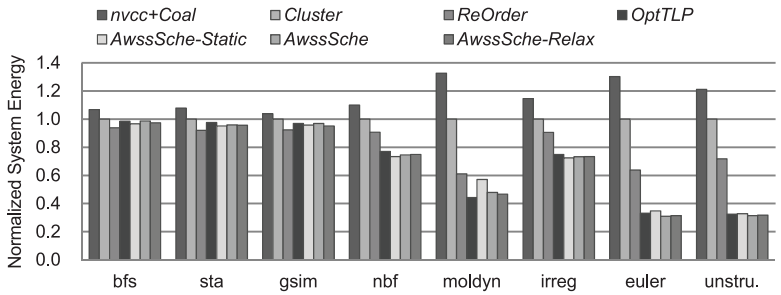

Fig. 10. The system energy comparison of various schemes. Numbers are normalized to the cluster scheme.

Finally, the energy consumption of different schemes is summarized in Figure 10. In general, Figure 10 shows a strong correlation between cache misses and system energy. As a result, by mitigating the cache contention, *AwssSche* reduces the system energy by 31.3% on average.

6.5. Sensitivity to Synchronization Cost

Different synchronization implementations could have different cost on the execution time. For the hardware-based implementation, recall that the cost of the synchronization function *syncthreads*() has been reported to take about 70 cycles per call [Wong et al. 2010]. On the other hand, this article also measures the cost of the software-based synchronization such as the kernel splitting. According to the measurements on the three NVIDIA GPGPUs (Tesla C1060, Tesla C2050, and Tesla K20), the cost of software-based implementation varies from 2k to 8k cycles. Knowing the large gap between hardware and software implementation, it is important to know how effectively the proposed thread scheduling scheme could perform in terms of different synchronization costs. This characteristic is known as the sensitivity to synchronization cost. This article defines this sensitivity on the basis of the invariance of the performance of a thread scheduling scheme when applied on architectures with different synchronization costs. This section uses different synchronization costs as the key parameter to stress this sensitivity.

Figure 11 shows the normalized execution cycle of the *AwssSche* and *AwssSche-Relax* schemes when the synchronization cost is increased from 50 to 200 cycles and from 2k to 8k cycles. Note that the other schemes do not issue any synchronization operation and thus are not shown in the figure. One can find that both the *AwssSche* and *AwssSche-Relax* schemes deliver have a very similar characteristic in terms of varied synchronization costs. Meanwhile, one can also find that the *AwssSche-Relax* scheme can slightly outperform the *AwssSche*. This is because *AwssSche-Relax* has the flexibility to search a larger solution space to achieve higher TLP and fewer synchronization operations. In summary, both *AwssSche* and *AwssSche-Relax* deliver good performance sensitivity with different synchronization costs.

6.6. Adaptability to Number of Cores

This section evaluates adaptbility to number of cores, defined as how effective a thread scheduling scheme is when being applied on architectures with different numbers of cores. This section uses different numbers of SMs as the key parameter to stress this adaptability experiment.

Figure 12 shows the adaptability when the architecture is configured to issue 8 to 32 SMs. Similar to the discussion in Section 6.2, when 8 SMs are used for the three EDA applications, the cache contention is very mild and the reorder scheme achieves

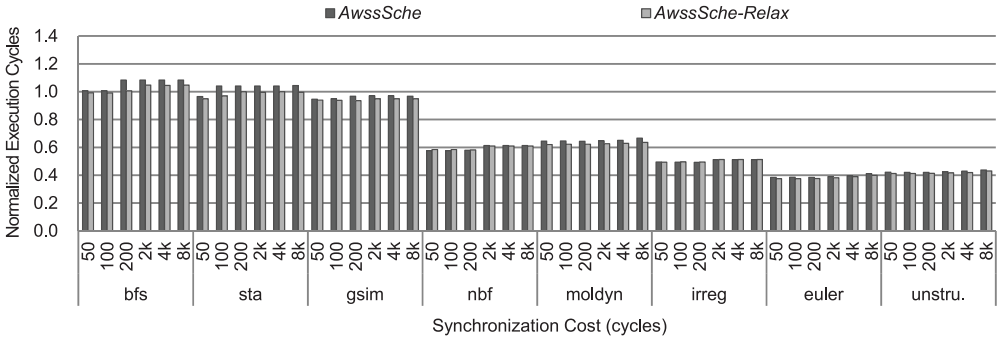

Fig. 11. The execution-time comparison of *AwssSche* and *AwssSche-Relax* schemes with different synchronization costs. Numbers are normalized to the cluster scheme.

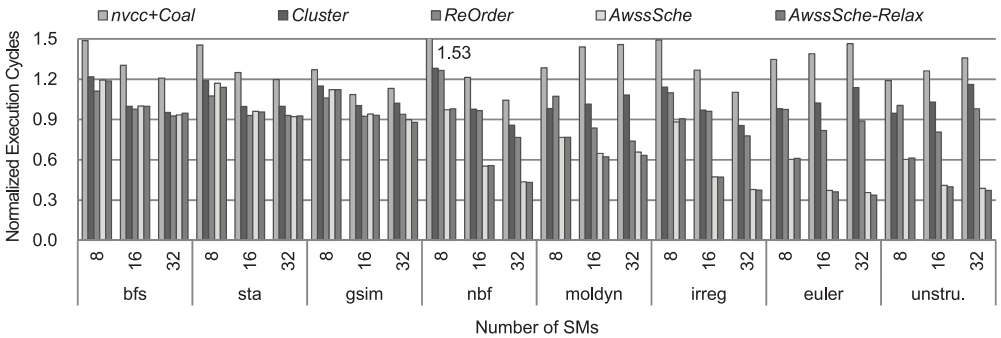

Fig. 12. The execution-time comparison of various of schemes with different numbers of SMs. Numbers are normalized to the cluster scheme with 15 SMs.

a slight improvement over *AwssSche* by further exploiting the data reuse. Another interesting observation is that, without the aggregate working-set size constraint, the *nvcc-Coal*, cluster, and reorder schemes can cause even worse performance when the number of SMs is increased. This is because when the number of SMs is increased, cache contention takes place and impacts the performance significantly. In contrast, by means of the cache contention constraint, *AwssSche* demonstrates better adaptability through avoiding the cache contention in SLLC.

6.7. Adaptability to Cache Capacity

This section further extends the adaptabilty evaluation to the SLLC capacity. This adaptability is defined as how effective a thread scheduling scheme is when applied on architectures with different SLLC capacities. This section uses SLLC capacity as the key parameter to stress such an adaptability experiment. Figure 13 shows the adaptability when the architecture is configured with 384k to 1536k of SLLC capacity. As shown in the figure, *AwssSche* behaves generally better than other schemes with different SLLC capacity. None the less, when 384k SLLC is used for the *bfs* and *sta* applications with very mild cache contention, the data reuse becomes more important and the reorder scheme achieves a slight improvement over *AwssSche* by further exploiting the data reuse. Besides, when 384k SLLC is used for the *moldyn* application, *AwssSche* tends to render too low TLP for avoiding cache contention. As a result, this causes slight performance degradation. *AwssSche-Relax* alleviates the performance degradation by rendering higher TLP.

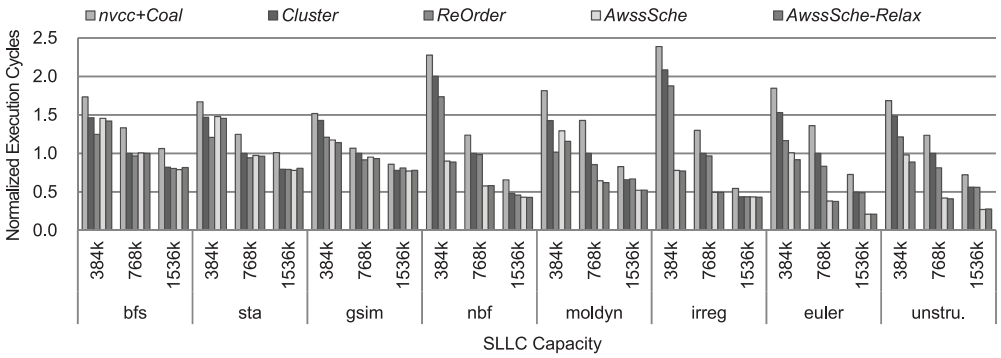

Fig. 13. The execution-time comparison of various schemes with different SLLC sizes. Numbers are normalized to the cluster scheme with a 768k SLLC.

Table V. Irregular Circuits and Unstructured Meshes

	Irregular Circuits				Unstructured Meshes			
Characteristics	*b17*	*b18*	*b19*	*b20*	*foil*	*mol1*	*mol2*	*auto*
Number of nodes	32,160	114,442	230,960	20,204	144,649	131,072	442,368	448,695
Number of edges	63,497	219,735	443,039	38,703	1,074,393	1,179,648	3,981,312	3,314,611

6.8. Scalability on Problem Size

This section evaluates the scalability on problem sizes. This article defines this scalability as how well a thread scheduling scheme can handle a problem when the problem size increases. This section uses different input files as the key parameter to stress this scalability experiment. Table V shows the statistics of the irregular circuits and unstructured meshes. The three EDA applications take the input files *b17* to *b20* from the ITC'99 circuit suite [Rogers et al. 2012], whereas the scientific applications take the input files *foil* to *auto* obtained from the COSMIC project [Han and Tseng 2006]. To observe the scalability with scaled problem sizes, in the following discussions, all the input files are sorted in ascending order of number of nodes and edges.

Figure 14 shows the scalability of different schemes. In the three EDA applications with mild cache contention, one can observe that the execution time of all the cache-contention-aware schemes cluster, reorder and *AwssSche* have very similar performance except in the *b19* dataset. In the *b19* dataset, there is greater potential of data reuse and the reorder scheme achieves a slight improvement over *AwssSche* by further exploiting the data reuse. However, it becomes more complicated in the other scientific applications in which cache contention takes place and impacts the performance significantly. In the scientific applications, one can observe that the execution time of the three schemes *nvcc-Coal*, cluster, and reorder grows extremely fast as the problem size increases. On the other hand, *AwssSche* and *AwssSche-Relax* achieve the best scalability by effectively constraining the aggregate working-set size to avoid cache contention in SLLC.

7. RELATED WORK

Optimizing shared last-level cache is remarkably critical to alleviate the memory bottleneck in modern Chip MultiProcessors (CMPs). For CMPs that integrate multiple conventional CPUs, some studies optimized the data reuse of the shared last-level cache by using thread scheduling techniques [Chen et al. 2007; Zhang et al. 2011b] and manual program transformations [Zhang et al. 2012]. GPGPUs are emerging as

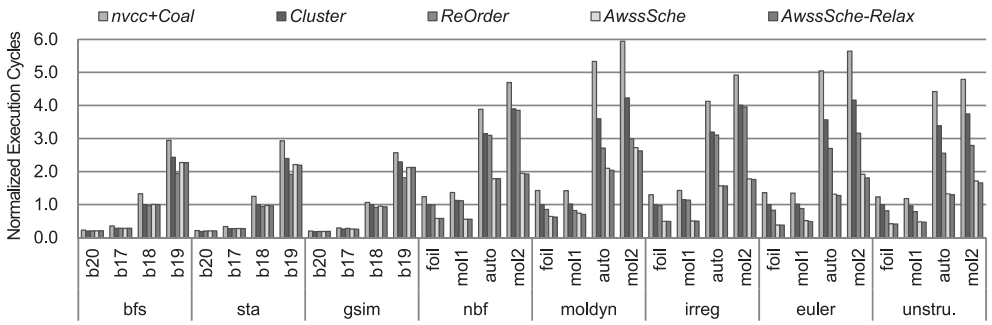

Fig. 14. The execution-time comparison of various schemes with different datasets. Numbers are normalized to the cluster scheme with *b18* and *foil* dataset.

a popular throughput-oriented architecture in modern CMP systems. For GPGPUs, most of the existing studies enhanced the performance by using the on-chip scratch-pad memory (also known as shared memory in NVIDIA's terminology) and the coalescing techniques. The related studies on handling regular memory access behavior include the compiler-based approaches Baskaran et al. [2008] and Yang et al. [2010], and auto-tuning [Ryoo et al. 2008]. To handle the irregular data accesses on GPGPUs, the authors of Deng et al. [2009] proposed a programming technique to use only the scratch-pad memory to manage the irregularity of data accesses. Different from relying on the scratch-pad memory, this article concentrates on the shared last-level cache in GPGPUs. In GPGPUs, data and computation reordering have been adopted to improve the coalescing [Wu et al. 2013; Zhang et al. 2011a]. Recently, a thread clustering technique had been proposed to consider both the coalescing and the data reuse in the shared L1 cache of GPGPUs [Kuo et al. 2012]. By combining the data and computation reordering techniques [Wu et al. 2013; Zhang et al. 2011a] and data reuse optimization techniques [Chen et al. 2007; Zhang et al. 2011b], this article implements the reorder scheme to exploit data reuse by adjusting the scheduling order of thread groups. Cache contention can be somehow mitigated while the data reuse is optimized, however, without an appropriate optimization, cache contention can still happen and impair the system performance even after applying these techniques. In order to address this requirement, recent works proposed hardware-based schedulers to dynamically manage the cache contention in an L1 cache. Some works tried to detect the contention of the L1 cache by the scheduler and scaled down the number of warps sharing the cache to alleviate the contention [Kayiran et al. 2012; Rogers et al. 2012]. Other works proposed schedulers to predict the memory footprint of warps and proactively prevented the cache contention in the L1 cache [Rogers et al. 2013]. Based on the consideration of cache contention, a further hardware-based scheduling technique was proposed to improve DRAM utilization [Jog et al. 2013]. Despite the hardware overhead induced in these works, this technique has shown its effectiveness at achieving appropriate thread-level parallelism and improving the system performance. In contrast to hardware-based approaches, this article concentrates on the software-based minimization of cache contention in the shared last-level cache. Furthermore, this work is orthogonal to the hardware-based approaches as the proposed aggregate working-set size constrained thread scheduling performs the scheduling at the cooperative-thread-array level while the previous works are at the warp level. As a result, aggregate working-set-size-constrained thread scheduling can be implemented together with any of the warp schedulers. To our best knowledge, this article is the first software-based approach that introduces the aggregate working-set-size-constraint to minimize the cache contention of the shared last-level cache in throughput processors.

8. CONCLUSIONS

This article has characterized and analyzed the performance impact of cache contention on the shared last-level cache of throughput processors. Based on the analyses and findings of cache contention and its performance pitfalls, this research formally formulates the aggregate working-set-size-constrained thread scheduling problem, which applies a constraint of aggregate working-set size on concurrent threads. With a proof of its NP-hardness, this article has adopted a series of algorithms for the thread scheduling problem and its variants. These algorithms are integrated into a generic framework for the further consideration of different architectural parameters and limitations. By applying the aggregate working-set size constraint, the proposed aggregate working-set-size-constrained thread scheduling successfully minimizes the cache contention and enhances the overall system performance on GPGPUs. The simulation results on NVIDIA's Fermi architecture have shown that the proposed thread scheduling scheme achieves an average of 46.8% reduction in cache misses and 31.8% execution-time enhancement over a thread clustering scheme. For applications with more threads and higher complexity, the execution-time improvement can reach up to 61.6%. When compared to a more sophisticated scheme that exploits the data reuse inherent in applications, the average improvement in execution time is 22.9% and up to 47.4% for applications with more complex workloads. Notably, the execution-time improvement of the proposed thread scheduling scheme is only 2.6% from an exhaustively searching scheme.

REFERENCES

Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing cuda workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. 163–174.

Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. 2008. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS'08)*. 225–234.

Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. 2007. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*.

Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. 1994. Communication optimizations for irregular scientific computations on distributed memory architectures. *J. Parallel Distrib. Comput.* 22, 462–478.

Yangdong Deng, Bo David Wang, and Shuai Mu. 2009. Taming irregular EDA applications on GPUs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'09)*. 539–546.

Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. 353–364.

Wu-Chun Feng and Shucai Xiao. 2010. To GPU synchronize or not GPU synchronize? In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'10)*. 3801–3804.

Michael R. Garey, Ronald L. Graham, and Jeffery D. Ullman. 1972. Worst-case analysis of memory allocation algorithms. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing (STOC'72)*. 143–150.

Michael Garland and David B. Kirk. 2010. Understanding throughput-oriented architectures. *Comm. ACM* 53, 58–66.

Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Proceedings of the Conference on Innovative Parallel Computing (InPar'12)*.

Hwansoo Han and Chau-Wen Tseng. 2006. Exploiting locality for irregular scientific codes. *IEEE Trans. Parallel Distrib. Syst.* 17, 606–618.

Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative thread array aware

scheduling techniques for improving GPGPU performance. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. 395–406.

Daniel R. Johnson, Matthew R. Johnson, John H. Kelm, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. 2011. Rigel: A 1,024-core single-chip accelerator architecture. *IEEE Micro* 31, 30–41.

Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. 2012. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. http://www.cse.psu.edu/~oik5019/docs/pdf/NMNL-PACT2013.pdf.

Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE Micro*. 31, 7–17.

Khronos. 2011. The opencl specification version 1.1. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

Kenneth L. Krause, Vincent Y. Shen, and Herb D. Schwetman. 1975. Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *J. ACM*. 22, 522–550.

Hsien-Kai Kuo, Kuan-Ting Chen, Bo-Cheng Charles Lai, and Jing-Yang Jou. 2012. Thread affinity mapping for irregular data access on shared cache GPGPU. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC'12)*. 659–664.

Bo-Cheng Charles Lai, Hsien-Kai Kuo, and Jing-Yang Jou. 2014. A cache hierarchy aware thread mapping methodology for GPGPUs. *IEEE Trans. Comput. PP,* 99, 1–1.

John Nickolls and William J. Dally. 2010. The GPU computing era. *IEEE Micro* 30, 56–69.

Nvidia. 2012a. NVIDIA kepler compute architecture whitepaper. http://www.nvidia.com/object/nvidia-kepler.html.

Nvidia. 2012b. NVIDIA cuda C programming guide 4.1. https://developer.nvidia.com/cuda-toolkit-41-archive/.

Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. 72–83.

Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. 99–110.

Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-Mei W. Hwu. 2008. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*. 195–204.

Jinuk Luke Shin, Kenway Tam, Dawei Huang, Bruce Petrick, Ha Pham, Changku Hwang, Hongping Li, Alan Smith, Timothy Johnson, Francis Schumacher, David Greenhill, Ana Sonia Leon, and Allan Strong. 2010. A 40nm 16-core 128-thread CMT SPARC SoC processor. In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC'10)*. 98–99.

Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. 2011. Fermi GF100 GPU architecture. *IEEE Micro* 31, 50–59.

Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'10)*. 235–246.

Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. 2013. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *Proceedings of the 18th ACM SIGPLAN Symposium Principles and Practice of Parallel Programming (PPoPP'13)*. 57–68.

Jian Yang and Joseph Y.-T. Leung. 2003. The ordered open-end bin-packing problem. *Oper. Res.* 51, 759–770.

Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. 86–97.

Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011a. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. 369–380.

Yuanrui Zhang, Mahmut Kandemir, and Taylan Yemliha. 2011b. Studying inter-core data reuse in multicores. In *Proceedings of the ACM SIGMETRICS joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'11)*. 25–36.

Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. 2012. The significance of CMP cache sharing on contemporary multithreaded applications. *IEEE Trans. Parallel Distrib. Syst.* 23, 367–374.

Received September 2013; revised May 2014; accepted June 2014