Exploring Cache Bypassing and Partitioning for Multi-Tasking on GPUs

Yun Liang
School of EECS, Peking University
ericlyun@pku.edu.cn

Xiuhong Li
School of EECS, Peking University
lixiuhong@pku.edu.cn

Xiaolong Xie
School of EECS, Peking University
xiexl_pku@pku.edu.cn

***Abstract*—Graphics Processing Units (GPUs) computing has become ubiquitous for embedded system, evidenced by its wide adoption for various general purpose applications. As more and more applications are accelerated by GPUs, multi-tasking scenario starts to emerge. Multi-tasking allows multiple applications to simultaneously execute on the same GPU and share the resource. This brings new challenges due to the contention among the different applications for the shared resources such as caches. However, the caches on GPUs are difficult to use. If used inappropriately, it may hurt the performance instead of improving it.**

In this paper, we propose to use cache partitioning together with cache bypassing as the shared cache management mechanism for multi-tasking on GPUs. The combined approach aims to reduce the interference among the tasks and preserve the locality for each task. However, the interplay among the cache partitioning and bypassing brings greater challenges. On one hand, the partitioned cache space to each task affects its cache bypassing decision. On the other hand, cache bypassing affects the cache capacity required for each task. To address this, we propose a two-step approach. First, we use cache partitioning to assign dedicated cache space to each task to reduce the interference among the tasks. During this process, we compare cache partitioning with coarse-grained cache bypassing. Then, we use fine-grained cache bypassing to selectively bypass certain data requests and threads for each task. We explore different cache partitioning and bypassing designs and demonstrate the potential benefits of this approach. Experiments using a wide range of applications demonstrate that our technique improves the overall system throughput by 52% on average compared to the default multi-tasking solution on GPUs.

***Index Terms*—GPU, Cache, Cache Partitioning, Cache Bypassing, Performance**

I. Introduction

GPUs are increasingly popular for embedded system computing as they can deliver orders of magnitude higher performance and energy efficiency than general purpose CPUs. For example, the system-on-chip (SoC) used by NVIDIA Tegra, Qualcomm Snapdragon, and Samsung Exynos series all integrate GPUs with CPUs [14], [16]. The presence of GPUs in SoC enables more and more general purpose and sophisticated applications on embedded system devices such as mobiles and tablets. As the number of applications ported onto GPUs continue to grow, multi-tasking scenario starts to emerge. For instance, a user of a tablet may execute face detection and object recognition simultaneously and both applications request for GPU acceleration. In reality, not all the GPU applications require full use of GPU resources [3]. Depending on the application's available parallelism and architecture's achievable memory bandwidth, GPU applications may saturate their performance with only a fraction of cores [3], [11]. Thus, multi-tasking that executes multiple applications on a single GPU has the potential to improve the resource utilization and overall performance.

However, multi-tasking on GPUs brings new challenges in the form of resource contention especially for caches. GPUs are inherently designed with small caches as most of the hardware area is devoted to computing resources. For example, on the latest NVIDIA TitanX GPU, the total L1 cache size across all the cores is only 1,152KB. Titan X can accommodate 49,152 concurrent threads in total, leading to 24 bytes cache capacity per thread. Therefore, the limited cache capacity of GPUs can be easily overwhelmed by the large number of threads, making GPU caches a system bottleneck and causing performance unpredictability [20]. Multi-tasking will aggravate this contention as the data locality of one application might be hampered by another through data eviction.

Cache bypassing techniques have been proposed for GPUs to alleviate the cache contention by allowing certain data requests to bypass the L1 cache [4], [5], [9], [20], [21]. Both instruction and thread level cache bypassing techniques have been proposed for a single application on GPUs [21]. On the other hand, cache partitioning techniques have been mainly used for managing the last level cache (LLC) of multicore architecture to reduce the cache interference by strictly partitioning the cache among different applications and assigning each application to its dedicated cache space [17].

To address the challenges and opportunities of cache for multi-tasking on GPUs, we propose to synergistically combine cache partitioning and bypassing. The synergistic interaction between cache bypassing and partitioning introduces new challenges. On one hand, cache partitioning assigns a portion of cache for each task and the allocated cache capacity affects the cache bypassing decision of each task. On the other hand, cache bypassing determines the data requests and threads that access or bypass the cache, which affects the required cache capacity of each task.

Our goal is to maximize the overall system throughput of multi-tasking by reducing the cache contention among the co-running tasks and preserving the data locality of each task. We design a two-step approach to achieve this goal. We first use cache partitioning to partition the cache space to co-running tasks. Our algorithm compares different partition

978-1-5386-3093-8/17/$31.00 ©2017 IEEE

choices with coarse-grained cache bypassing (bypass-all). This is essential for GPU as the caches in GPU architecture may hurt the performance due to excessive cache contentions. For the determined cache partition of each task, we use fine-grained cache bypassing to selectively bypass certain data requests and threads.

We propose both static and dynamic cache partitioning approaches for multi-tasking on GPUs. For static approach, we first characterize each task on how the performance varies with different cache size in single-program execution mode and derive the overall system throughput in multi-program execution mode. For dynamic approach, we propose to dynamically adjust the cache partition by accommodating the task phase changing behavior and sensitivity to different inputs. For cache bypassing, we use instruction and thread level bypassing. Finally, we devise different designs by combining cache partitioning and bypassing.

To the best of our knowledge, this is the first work that synergistically uses cache bypassing and partitioning for multi-tasking on GPUs for performance optimization. This paper makes the following contributions.

- We propose a synergistic cache bypassing and partitioning approach for multi-tasking on GPUs to maximize the overall system throughput.
- We propose static and dynamic cache partitioning designs for multi-tasking on GPUs. We explore different design choices by combining cache partitioning and bypassing schemes.
- We perform rigorous experiment validation using multiple applications with different characteristics on different GPU architectures.

Experiments using a wide range of multi-tasking workloads demonstrate that by exploring different cache bypassing and partitioning designs, our approach can improve the overall system throughput by 52% on average compared to the default multi-tasking solution.

II. GPU Architecture Background

Figure 1 presents a GPU architecture in general. GPUs are single instruction multiple data (SIMD) architectures. When a GPU task (kernel) is launched, GPU spawns hundreds or thousands of threads using the same set of instructions. The threads are further divided into cooperative groups, named thread blocks. Threads within the same thread block can synchronize and exchange data during execution. One GPU is composed of multiple *Stream Multiprocessors* (SMs) and SMs are connected with shared off-chip L2 cache and device memory via interconnection network. Each SM, which is composed of multiple executing units, register file, L1 data cache, shared memory, and others resources, is able to execute multiple thread blocks either from the same or different applications concurrently.

GPU vendors have enabled multi-tasking (concurrent kernel execution) to improve the resource utilization since the Fermi architecture. Fermi architecture supports concurrent kernel execution from the same application, and the latest architectures

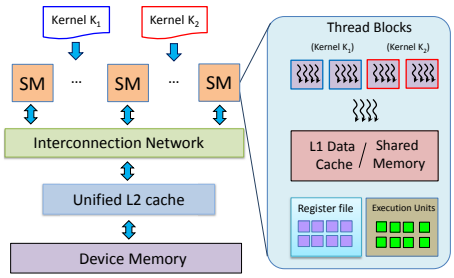


Fig. 1: GPU architecture.

(Kepler, Maxwell, and Pascal) improve Fermi by introducing the Hyper-Q feature, which maintains multiple independent kernel queues to concurrently execute independent kernels. Recently, NVIDIA propose multi-process service (MPS) to utilize the Hyper-Q feature in the software layer, which allows kernels from different applications to execute concurrently on the same GPU. However, the default multi-tasking solution on all the GPU architectures allow multiple applications to share the cache, which is highly susceptible to cache thrashing. In this work, we resort to cache partitioning and bypassing to solve the problem. Similar to the default multi-tasking and prior work [10], we allow multiple tasks to co-execute on the same SM for resource utilization efficiency.

Each SM on GPU is associated with its private L1 cache and all the thread blocks that are scheduled on the same SM share the L1 cache. A cache memory in general is defined in terms of three major parameters: *block or line size* L, *number of sets* S, and *associativity* W. The block or line size determines the unit of transfer between the main memory and the cache. A cache is divided into S sets. Each cache set, in turn, is divided into W cache blocks, where W is the associativity of the cache. Now the cache size is defined as $(L \times S \times W)$. In this work, we use the widely adopted way partitioning [17] for GPU cache. In way partitioning, particular ways of a set associative cache are selected and these ways are assigned to a task for all the cache sets as shown in Figure 2.

In this work, we configure the L1 cache bypassing in either coarse-grained or fine-grained manner. Since Fermi architectures, NVIDIA GPUs provide interfaces to explicitly control the L1 cache access or bypass for global load instructions. In a coarse-grained manner, all the global load instructions and threads are bypassed. In a fine-grained manner, we configure cache access or bypass at instruction and thread block level [21]. Finally, currently, on all the NVIDIA GPUs including Fermi, Kepler, Maxwell, etc, L1 caches in different SMs are not coherent with each other. The L2 cache is shared by and coherent across all the SMs on the chip.

III. Cache Management

A. Overview

GPU applications tend to launch a large number of threads and these threads often issue large amounts of memory requests within a short period, leading to high L1 cache contention. Despite that GPUs employ fast context switch to

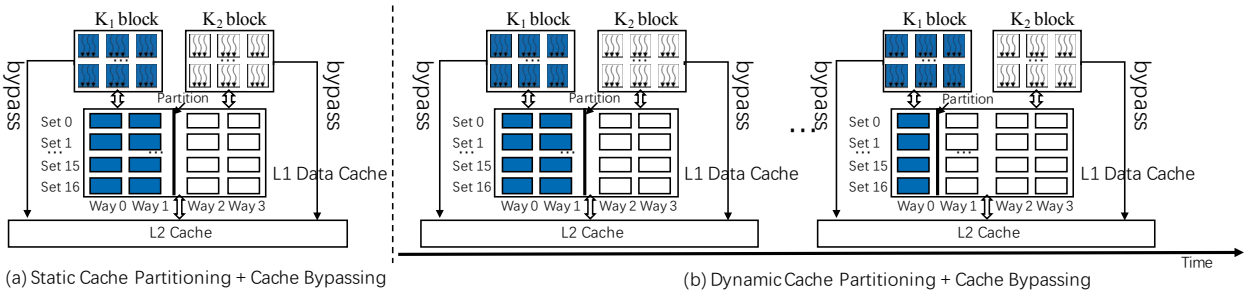


Fig. 2: Different design choices.

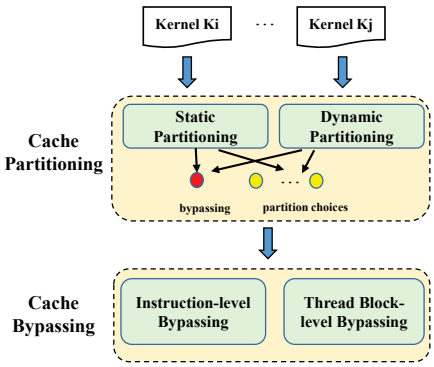


Fig. 3: Overview of the two-step approach.

hide the long memory latency, the cache contention remains a performance bottleneck [5]. The problem gets even worse in the multi-tasking scenario on GPUs as multiple tasks or kernels will compete for the cache resource. Prior work has shown that the parallelism (e.g. number of concurrent thread blocks of each task on an SM) composition of multiple tasks will also impact the performance [10]. In this work, we concentrate on the cache management technique for multi-tasking and will compare with [10] in the experiment.

The interplay between cache bypassing and partitioning in multi-tasking setting provides unique design choices and opportunities for us to solve the cache contention problem. We devise a two-step approach as shown in Figure 3. We first use cache partitioning to resolve the cache interference among different tasks. When we explore different partitioning choices in the design space, we compare with coarse-grained bypassing (bypass-all for all instructions and threads). This is crucial because for certain tasks cache may hurt the performance due to memory congestion and pipeline stall [21]. Cache partitioning assigns dedicated cache space to each individual co-running task. Then, for each task, we use fine-grained bypassing to select the instructions and thread blocks to bypass the cache for further performance improvement.

Finally, we devise different design choices by combining cache partitioning and bypassing schemes. We propose both static and dynamic cache partitioning approaches in tandem with cache bypassing. For static approach, as shown in Figure 2 (a), the cache partition is determined initially before the kernels start and remains unchanged throughout the kernel execution. Figure 2 (b) shows the dynamic cache bypassing, where the cache partition might change dynamically at run-time. Static approach makes decision statically based on the kernel characterization. In contrast, dynamic approach can adjust the partition to different task phase and input behavior but will incur learning overhead at run-time.

Figure 2 presents different design choices of this combined approach for a two-kernel case, where kernels K_1 and K_2 are executed on the same SM and share the L1 data cache. For static and dynamic cache partitioning design, cache bypassing is employed for each kernel to bypass the L1 cache and access the L2 cache directly for certain data requests and threads.

B. Cache Partitioning

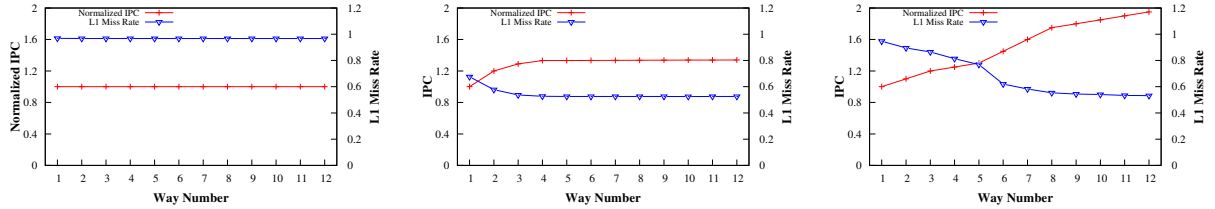
1) Kernel Characterization: We first classify the GPU kernels into three types based on how the performance varies as the cache size (cache ways) increases in single-program execution mode. We use IPC (instruction per clock) to measure the performance. Note that we use cache way partition, where the cache set and block size remain unchanged.

- *Compute Intensive: (abbr. C)* The performance nearly stays unchanged as the number of cache way increases. Figure 4a uses the kernel *SAD* in Table I for illustration. Since there exists no data locality, neither bypassing nor accessing cache is useful for the kernels in this type.
- *Memory Intensive-Saturate: (abbr. S)* The performance first increases and then saturates as the number of cache way increases. Figure 4b uses the kernel *BP* in Table I for illustration. For the kernels in this type, memory pressure is alleviated as the number of cache way increases.
- *Memory Intensive-Increase: (abbr. I)* The performance continues to increase as the number of cache way increases. Figure 4c uses the kernel *SC* in Table I for illustration. For the kernels in this type, they have good locality and prefer large caches.

Next, we will present the details of our static and dynamic cache partitioning designs.

2) Static Cache Partitioning: Given a set of kernels that simultaneously execute on the GPU, $\mathcal{K} = \{k_1, k_2, \ldots, k_N\}$, we use the overall system throughput ($STP_{\mathcal{K}}$) as the optimization objective,

$$STP_{\mathcal{K}} = \sum_{i=1}^{N} \frac{IPC^{mp}_{k_i,c_i}}{IPC^{sp}_{k_i}} \quad (1)$$



(a) *Compute Intensive (Type: C).* (b) *Memory Intensive-Saturating (Type: S).* (c) *Memory Intensive-Increasing (Type: I).*

Fig. 4: Kernel Characterization. (a), (b) and (c) uses kernels *SAD*, *BP*, and *SC*, respectively

where $IPC^{sp}_{k_i}$ denotes the performance (IPC) of the kernel k_i in single-program execution mode, $IPC^{mp}_{k_i,c_i}$ denotes the performance of the kernel k_i with c_i allocated cache ways in multi-program execution mode. In single-program execution mode, each kernel will exclusively use the entire cache. Note that our technique in the following is not limited to the system throughput optimization, it can also be used for other multi-tasking evaluation criteria such as fairness [22].

Then, we formulate the optimization problem of static cache partitioning as follows,

Problem 1: Given a set of kernels $\mathcal{K} = \{k_1, k_2, \ldots, k_N\}$, the static cache partitioning problem aims to partition the L1 cache on each SM into N disjoint subsets $(\{C[1], C[2], ..., C[N]\})$ of cache ways so as to maximize the system throughput with subject to $\sum_{i=1}^{N} C[i] \leq W$, where W is the number of L1 cache ways.

For the kernels that belong to *Compute Intensive* type, we will let them completely bypass the cache since they will not benefit from the cache. For the kernels of the other two types, intuitively using more cache ways will be no worse than using less cache ways. However, it gets complicated when cache bypassing is considered as shown in Figure 5. In Figure 5, zero cache ways is equivalent to coarse-grained bypassing, which bypass all the data requests to L2 cache. For example, for kernel *stream cluster*, cache bypassing achieves better performance than accessing the cache. This is because the massive thread parallelism causes serious memory congestion and pipeline stall overhead for it when L1 cache is used [21]. If the performance gain from exploiting the data locality by using cache can not offset the pipeline stall overhead, cache bypassing will be a better choice.

For each kernel k_i, we use $IPC^{sp}_{k_i,bypass}$ and $IPC^{sp}_{k_i,c_i}$ to denote its IPC when cache is bypassed and allocated c_i cache ways in single-program execution mode, respectively. Then, we define $bypass(k_i)$ as follows,

$$bypass(k_i) = \begin{cases} true & IPC^{sp}_{k_i,bypass} \geq IPC^{sp}_{k_i,1} \\ false & \text{otherwise} \end{cases}$$

For kernel k_i, if $bypass(k_i)$ is true, we will evaluate both cache bypassing and cache accessing for kernel k_i; otherwise, we will only consider cache accessing for kernel k_i.

Algorithm 1 presents our detailed algorithm to Problem 1. We define $S^{candidate}_{bypass}$ as the set of kernels that are candidates for cache bypassing. $S^{candidate}_{bypass}$ contains all the kernels of which $bypass(k_i)$ is true (line 3-6). For each kernel in the

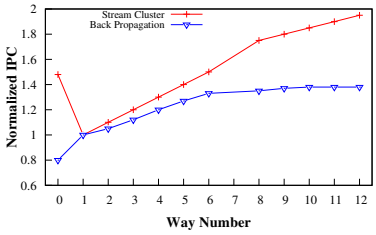


Fig. 5: Cache bypassing vs accessing.

$S^{candidate}_{bypass}$, we consider cache bypassing and accessing. Then, we enumerate all the possible cases for them (line 9-26). Each case is represented using a M-bit vector (line 12).

In Algorithm 1, we use set S_{access} to represent the kernels which will access the cache, and set S_{bypass} to represent the kernels which will bypass the cache. If we let a cache bypassing candidate to access the cache, then it is assigned with 1 cache way initially (line 18). Then, for each kernel in set S_{access}, its performance exhibits a non-decreasing trend as the cache ways increases as shown by Figure 4b and Figure 4c. Then, we rely on the separable convex resource allocation problem [18] to derive the optimal solution.

Function *Partition* (line 27-32) gives the implementation details. We use $STP[i][x]$ to represent the additional weighted system throughput for kernel k_i when the assigned cache ways increases from x to $x+1$. $STP[i][x]$ is obtained as a by-product during the kernel characterization step in Section III-B1. Then, each time we increase $c[j]$ by one, where j is the index for which $STP[j][c[j]]$ is the largest, and then repeat the above operation, until all the cache ways have been used. At last, function *Partition* can yield a cache partition, then we compare the overall system throughput of this cache partition with that of the current best cache partition and update if necessary. For simplicity, we use $IPC^{sp}_{k_i,c_i}$ in single-program mode to approximate the $IPC^{mp}_{k_i,c_i}$. In other words, we mainly focus on the contention from L1 cache, but ignore the contention from L2 cache and interconnect, etc. The time complexity of Algorithm 1 is $W * 2^M$.

3) Dynamic Cache Partitioning: For static cache partitioning scheme, the cache partition for each task is decided at the beginning and keeps unchanged throughout the execution. More importantly, the static scheme uses the average statistics (e.g, IPC, type) as guided metrics which are obtained by profiling the representative inputs. However, in reality, the application's behavior might change during runtime for the same input and across different inputs, making static solution not

Algorithm 1: Static cache partitioning algorithm

Input : $\mathcal{K}$, $STP[1..N][1..N]$
Output: $C_{opt}[1..N]$
$C_{opt}[] = 0$;
$S_{bypass}^{candidate} = \emptyset$;
foreach *kernel* $k_i \in \mathcal{K}$ **do**
  if $bypass(k_i)$ *is true* **then**
    $S_{bypass}^{candidate} = S_{bypass}^{candidate} \bigcup \{k_i\}$;
  end
end
$M = S_{bypass}^{candidate}.size()$;
for $(i = 0; i < 2^M; i++)$ **do**
  $C[] = 0$;
  $S_{bypass} = S_{bypass}^{candidate}$;
  let B be the M-bit bitset of i;
  for $(j = 0; j < M; j++)$ **do**
    if *B[j] = 1* **then**
      C[j] = 0;
    else
      $S_{bypass} = S_{bypass} \setminus \{k_j\}$;
      C[j] = 1;
    end
  end
  $S_{access} = \mathcal{K} \setminus S_{bypass}$;
  Partition(S_{access}, C[]);
  if $STP_{\mathcal{K}}(C[]) > STP_{\mathcal{K}}(C_{opt}[])$ **then**
    $C_{opt}[] = C[]$;
  end
end
function *Partition(S, C[])*
  repeat
    Increase $C[k_j]$ by one, where kernel k_j has the largest $STP[k_j][C[k_j]]$ in Set S;
  until $sum(C[1..N]) \geq W$;
  return $C[]$;
end

optimal. Here, we devise a hardware-based dynamic scheme that can adjust the cache partitioning according to program behavior at run-time.

For the kernel set $\mathcal{K} = \{k_1, k_2, \ldots, k_N\}$, we select N SMs to perform hardware-based profiling. We call the N SMs as *Profiling SMs* and the remaining SMs as *Followed SMs*. We divide the task execution into multiple sampling period. In general, in the current sampling period, the *Profiling SMs* are mainly used to predict the best cache partition in the next sampling period and the *Followed SMs* will use the predicted best cache partition in the next sampling period. The detailed workflow of our dynamic scheme is as follows,

1) Initially, we use *Even* cache partition on all SMs including *Profiling SMs* and *Followed SMs*. For each SM, we evenly partition the cache ways among all the kernels.
2) At the beginning of each sampling period, we use each *Profiling SM* to represent a different cache partition adjusting trend. More clearly, based on the cache partition of the *Followed SMs* in the previous sampling period, we increment the number of cache ways of the i^{th} kernel by one for the *Profiling* SM_i, and randomly reduce one cache way from the other kernels on SM_i .
3) At the end of each sampling period, we collect the IPC of all the SMs. Then, the comparator will compare the system throughput among the *Profiling SMs* and *Followed SMs*, and pick the SM with the maximum system throughput as the winner.
4) Update the cache partition of the *Followed SMs* using the winner's configuration.
5) Repeat Step 2-5 until the end of the execution.

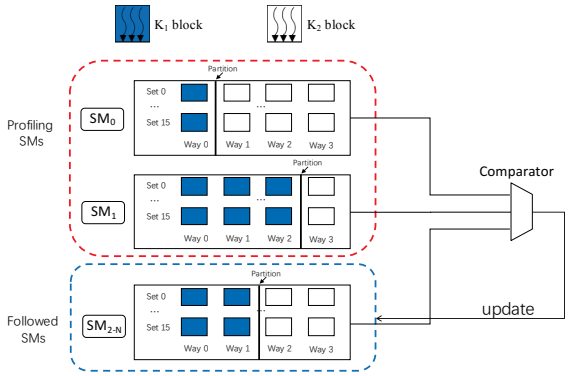


Fig. 6: Dynamic cache partitioning scheme.

Let us assume that Tb_i thread blocks of kernel k_i are accommodated within an SM. Then, we define the length of the sampling period as the time which all the SMs have at least finished Tb_i thread blocks for each kernel k_i. This ensures that each task makes sufficient progress within one sampling period. Figure 6 shows an illustration of the dynamic cache partition scheme for a two-kernel case. The L1 cache consists of four ways. Figure 6 shows the second sampling period, where the *Followed SMs* use even cache partition and *Profiling SMs* SM_0 and SM_1 give one more cache way to kernels k_1 and k_2, respectively.

Hardware Implementation Overhead. Our framework requires very small area for hardware implementation. For each SM, we only need 2 32-bit registers, one for storing the current cache partition scheme and the other one for storing the IPC at the end of each sampling period. Furthermore, we need a comparator to compare the system throughput.

C. Cache Bypassing

The main purpose of cache partitioning is to solve the cache contention among multiple GPU kernels, however, the cache contention problem of each single kernel remains unsolved. More importantly, after cache partitioning, the cache contention problem may be aggravated as the cache size of each kernel shrinks. Cache bypassing on GPUs allows cores to bypass L1 cache and access lower-level cache directly, has been demonstrated to be effective to mitigate cache contention on GPUs [4], [21]. In this section, we explore fine-grained cache bypassing for better performance.

Various cache bypassing techniques for GPUs have been discussed. For example, instruction level cache bypassing classifies memory access instructions at compile-time and bypass the L1 cache for memory instructions that do not benefit from cache [9], [20]. Other bypassing techniques make the bypassing decision at run-time according to the monitored cache performance [4], [5], [20]. We adopt the technique proposed by Xie et al. [21] that combines instruction and thread block level cache bypassing. The instruction level cache bypassing identifies those memory instructions that have strong cache preference, either prefer using cache or bypassing

cache. For the rest memory instructions, their cache preference may vary according to the run-time information. A thread block level cache bypassing is carried out to tune the number of thread blocks that use or bypass cache adaptively.

More clearly, the instruction level cache bypassing uses profiling to categorize the instructions. It classifies memory instructions that have low hit rate as no locality instructions and do not allow them to use cache. Instructions that have high hit rate are allowed to always use cache. The cache behavior of the rest memory instructions are adjusted by thread block level cache bypassing at run-time. The basic idea is, at run-time, when cache contention is observed, it increases the number of thread blocks that bypass the L1 cache to mitigate the contention. Otherwise, it decreases the number of threads blocks that bypass the L1 cache, i.e., allows more thread blocks to use the cache, to exploit the data localities. To this, we use the *CHSS* metric designed by Xie et al. [21] as the cache performance indicator. *CHSS* is defined as

$$CHSS = \frac{Hits \cdot L2\_Latency}{Stall \cdot WarpCount} \quad (2)$$

where $Hits$ is the number of cache hits during the sampling period, $L2\_Latency$ is the L2 cache access latency, $Stall$ is the number of pipeline stalls caused by accessing the cache, and *WarpCount* is the number of active warp.

Thread block level cache bypassing works as follows. At run-time, it monitors the $CHSS$ during a fixed time interval. When a thread block retires and a new thread block is issued to the GPU, it checks $CHSS$ to determine the cache or bypass behavior of the new thread block. The length of the time interval is set as the time span of a thread block. By doing that, the thread block level cache bypassing can adjust the cache or bypass behavior of GPU tasks adaptively. Similar to prior work [21], our cache bypassing will not affect the data consistency and cache coherence. This is because it works at thread block granularity. Given a thread block, all the threads in it have the same memory behavior (cache or bypass).

IV. Experiment Evaluation

We implement our techniques based on GPGPU-Sim 3.3.2. Both Fermi and Kepler-like architectures are evaluated. The detailed configurations are shown in Table II. We evaluate our techniques using 10 representative benchmarks shown in Table I, among which there are 6 memory intensive benchmarks and 4 compute intensive benchmarks. With the 10 benchmarks, we create 39 two-kernel workloads, among of which 15 workloads consist of two memory intensive benchmarks and 24 workloads consist of a compute intensive and a memory intensive benchmarks. In our evaluation, we do not consider the combination of two compute intensive workloads as neither of them prefers to use cache. The static cache partitioning requires profiling inputs to characterize the kernels. We use different inputs for profiling and evaluation purpose. The inputs used in the experiments are described in Table II.

We perform evaluation from the following four aspects. First, we present the results of static and dynamic cache partitioning alone. Second, we explore cache partitioning together with cache bypassing . Third, we show the scalability results as the cache size and the number of kernels increases. Finally, we compare with the state-of-the-art techniques.

TABLE I: Kernel description.

Application Name	Profiling Input (#Inst)	Evaluation Input (#Inst)	Type
Back Propagation(BP) [2]	36M	72M	*S*
Heart Wall(HW) [2]	52M	52M	*S*
Breadth First Search(BFS) [2]	0.9M	41M	*S*
Lattice-Boltzman Method [1] (LBM)	0.56B	0.56B	*S*
K-means (KM) [2]	0.15B	0.15B	*I*
Stream Cluster (SC) [2]	77M	0.15B	*I*
HotSpot (HS) [2]	0.44B	0.11B	*C*
Sum of Absolute Differences (SAD) [1]	5.6M	0.45B	*C*
3-D Stencil Operation (STENCIL) [1]	27M	91M	*C*
Cutoff Coulombic Potential (CUTCP) [1]	0.15B	0.15B	*C*

TABLE II: GPGPU-Sim configuration.

	Fermi	Kepler
# Compute Units (SM)	15	15
SM configuration	32 cores, 700MHz	
Threads per SM	1536	2048
Warps Per SM	48	64
Warp Scheduler	2 warp schedulers per SM, GTO policy	
32-bit Registers/SM	32768	65536
L1 Data Cache	16KB, 32-set, 4-way, cache line (128B)	
L2 Unified Cache	768 KB, 700 MHz, 64-set, 8-way	

A. Performance Results

Figure 7 presents the performance results of cache partitioning alone on Fermi architecture. The system throughput is normalized to the default multi-tasking solution without cache management techniques. On average, static cache partitioning and dynamic cache partitioning can improve the system throughput by 42% and 33%, respectively. For most of the workloads, static cache partitioning shows obvious advantage over dynamic cache partitioning, especially for the *S_S* and *S_I* workloads. It is because static scheme uses accurate profiling information while the dynamic scheme often takes several sampling periods to converge to a stable solution. For *S_C* and *I_C* workloads, we find that the gap between static and dynamic partitioning is relatively smaller than other workloads. For these workloads, the static partitioning will directly bypass the *Compute Intensive* kernels. It is easy for dynamic partitioning to converge to a stable partition (i.e. bypassing the *Compute Intensive* kernels) on these workloads as cache size has negligible effect for *Compute Intensive* kernels. The dynamic cache partitioning can also outperform the static cache partitioning for certain workloads, such as *SC_HS* and *HW_SC*. For these cases, kernels *SC* and *HW* exhibit very obvious phase changing behavior and are sensitive to different inputs. Therefore, dynamic partition scheme turns out to be a better choice.

Figure 8 shows the results of combined cache partitioning and bypassing on Fermi architecture. On average, after incorporating with cache bypassing, the performance speedup of static partitioning is increased from 42% to 52%, and the performance speedup of dynamic partitioning is improved from 33% to 45%. In multi-tasking scenario, after cache partitioning, the cache capacity assigned to each kernel becomes

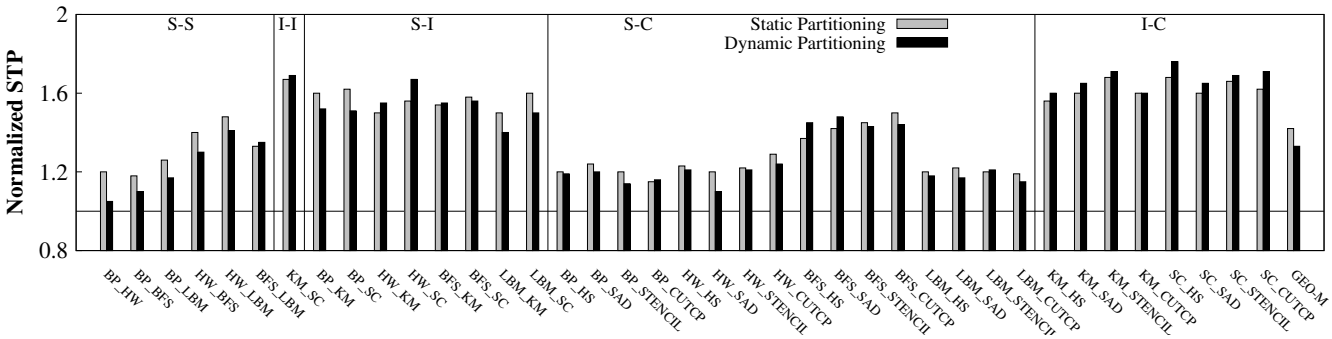


Fig. 7: Comparison of static and dynamic cache partitioning on Fermi-architecture

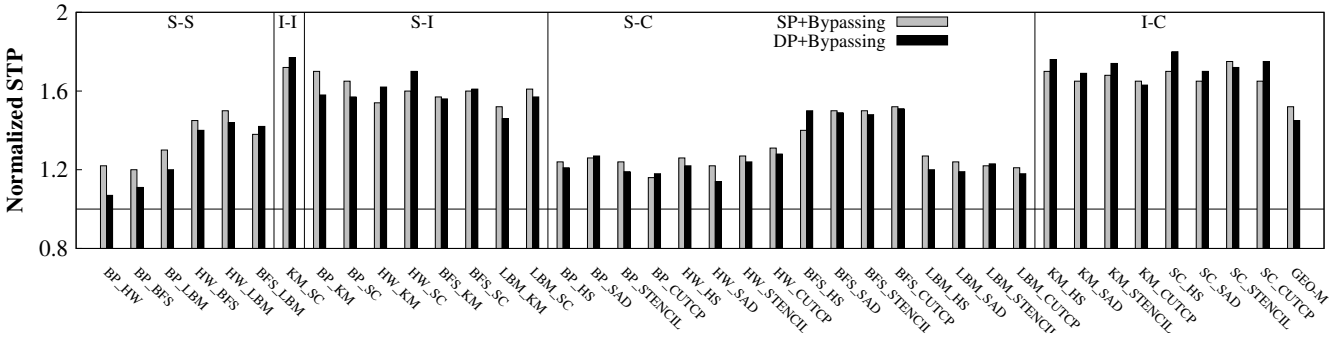


Fig. 8: Cache partitioning together with cache bypassing on Fermi-architecture.

smaller, leading to more serious cache contention problem. Thus, cache bypassing helps to further improve the performance. The system throughput is improved as a mixed effect of cache miss rate reduction and memory pipeline stall reduction. On average, our combined approach reduces L1 cache miss rate and pipeline stall by 16.6% and 21.3%, respectively.

Kepler Architecture. On Kepler architecture, our static and dynamic partitioning improve the system throughput by 47% and 36%, respectively. The combined approach further improves the performance to 55% and 46%. Therefore, our techniques are applicable to different GPU architectures.

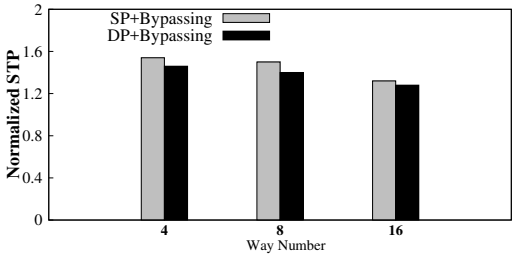


Fig. 9: Results of different cache configurations.

B. Scalability

Figure 9 presents the results as the number of cache ways increases. We keep the number of cache set and block size constant and thus the cache size will increase with the number of cache ways. The results demonstrate that considerable performance improvement is achieved for different cache settings. As we expect, the performance improvement will become smaller when the number of cache way or cache size increases. This is because the cache contention is reduced as the cache size increases. However, GPU architectures consistently use small caches under its massive parallelism design principle. Therefore, we need sophisticated cache optimization techniques to reduce the cache contention.

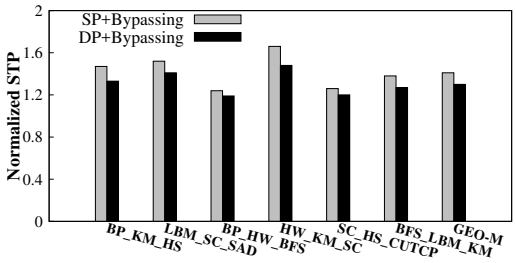


Fig. 10: Results of three kernel cases.

Figure 10 presents the results for three-kernel case. Due to the space limitation, we only show the results of six three-kernel workloads. Our combined approach improves the system throughput by 47% on average for three-kernel workloads.

C. Comparison

Li and Liang present an efficient multi-tasking framework on GPUs [10]. Their techniques employ a thread block modulation technique to alleviate the cache contention. They mainly

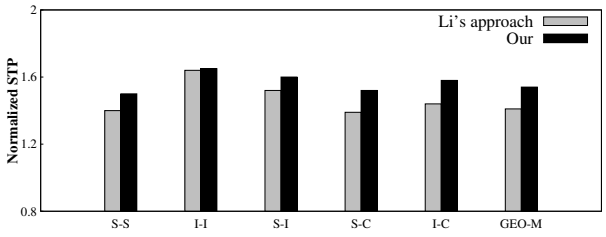


Fig. 11: Comparison with the state-of-the-art.

focus on the thread parallelism optimization, but ignore the cache optimizations. Figure 11 compares our solution with them for different types of workloads. Our approach improves the system throughput by 52% on average, while Li's approach improves by 41% on Fermi architecture.

V. Related Work

Cache Partitioning. Cache Partitioning has been demonstrated to be effective in mitigating the cache interference between parallel applications on CMPs [6], [17], [18]. It divides the cache into multiple partitions and during the execution, applications are not allowed to access the same partition simultaneously. By doing this, the cache interference between concurrent applications is avoided. Recently, cache partitioning is employed on GPUs to help to minimize the progress disparity of threads for single task case [8]. However, how to solve the cache contention in multi-tasking has not been explored on GPUs.

Cache Bypassing. Given the limited cache size and large amount of running threads, GPUs are facing serious cache contention problem [5]. Cache bypassing, that allows the GPU to bypass L1 cache for some of the memory accesses, has been widely discussed for the single task case. Both compile-time [5], [20] and run-time [5], [21] solutions are proposed. It has also been combined with other techniques, e.g. thread throttling, for better performance [4]. However, none of the existing works discusses cache bypassing in the case of multi-tasking. In this paper, we combine cache bypassing with cache partition and demonstrate that it is an effective technique in multi-tasking scenario.

GPU Multitasking. With more applications are now accelerated on GPUs, the need for efficient multi-tasking management on GPUs is necessary. Different multi-tasking solutions including preemptive multi-tasking [12], [15], [19], spatial multi-tasking [3], [7], [10], and temporal multi-tasking [11], [13] are proposed. In this paper, we focus on the spatial multi-tasking, where multiple GPU kernels are allowed to share one GPU simultaneously. When multiple kernels share the resources within one SM, they will compete for resources such as caches. However, none of the prior works has addressed this problem. In this paper, we solve this problem using a combined cache partition and cache bypassing design.

VI. Conclusion

Thanks to the tremendous computational power of GPUs, a wide range of applications have been ported to GPUs for performance acceleration. This leads to multi-tasking demands on GPUs. In this paper, we explore cache partitioning and bypassing together as cache optimization techniques for multi-tasking on GPUs to alleviate the cache contention and improve the overall system throughput. We first develop static and dynamic cache partitioning techniques. Then, we use instruction and thread block level cache bypassing to further improve the performance. By cooperating with the cache bypassing and cache partitioning, we can improve the system throughput by 52% on average.

Acknowledgment

This work is partially supported by the National Science Foundation China (No. 61672048). The corresponding author of this paper is Yun Liang (Email: ericlyun@pku.edu.cn).

References

[1] Parboil Benchmark Suite. http://impact.crhc.illinois.edu/Parboil/parboil.aspx.
[2] Rodinia Benchmark Suite. http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/.
[3] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for gpgpu spatial multitasking. In *HPCA*, 2012.
[4] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu. Adaptive cache management for energy-efficient GPU computing. In *MICRO*, 2014.
[5] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *HPCA*, 2014.
[6] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.
[7] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In *HPCA*, 2014.
[8] S.-Y. Lee, A. Arunkumar, and C.-J. Wu. CAWA: coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. In *ISCA*, 2015.
[9] A. Li, G.-J. van den Braak, A. Kumar, and H. Corporaal. Adaptive and transparent cache bypassing for GPUs. In *SC*, 2015.
[10] X. Li and Y. Liang. Efficient kernel management on GPUs. In *DATE*, 2016.
[11] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen. Efficient GPU spatial-temporal multitasking. *IEEE Transactions on Parallel and Distributed Systems*, 26(3):748–760, March 2015.
[12] Z. Lin, L. Nyland, and H. Zhou. Enabling efficient preemption for SIMT architectures with lightweight context switching. In *SC*, 2016.
[13] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *ASPLOS*, 2013.
[14] J.-G. Park, N. Dutt, H. Kim, and S.-S. Lim. HiCAP: Hierarchical FSM-based dynamic integrated CPU-GPU frequency capping governor for energy-efficient mobile gaming. In *ISLPED*, 2016.
[15] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. In *ASPLOS*, 2015.
[16] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. Integrated CPU-GPU power management for 3D mobile games. In *DAC*, 2014.
[17] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.
[18] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068, Sep 1992.
[19] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *ISCA*, 2014.
[20] X. Xie, Y. Liang, G. Sun, and D. Chen. An efficient compiler framework for cache bypassing on GPUs. In *ICCAD*, 2013.
[21] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. Coordinated static and dynamic cache bypassing for GPUs. In *HPCA*, 2015.
[22] S. M. Zahedi and B. C. Lee. Ref: Resource elasticity fairness with sharing incentives for multiprocessors. In *ASPLOS*, 2014.