Coordinated Static and Dynamic Cache Bypassing for GPUs

Xiaolong Xie[1], Yun Liang[1], Yu Wang[2], Guangyu Sun[1], and Tao Wang[1]

[1]*Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, China*
{xiexl_pku, ericlyun, gsun, wangtao}@pku.edu.cn
[2]*Tsinghua National Laboratory for Information Science and Technology, Department of EE, Tsinghua University, China*
yu-wang@mail.tsinghua.edu.cn

Abstract

The massive parallel architecture enables graphics processing units (GPUs) to boost performance for a wide range of applications. Initially, GPUs only employ scratchpad memory as on-chip memory. Recently, to broaden the scope of applications that can be accelerated by GPUs, GPU vendors have used caches in conjunction with scratchpad memory as on-chip memory in the new generations of GPUs. Unfortunately, GPU caches face many performance challenges that arise due to excessive thread contention for cache resource. Cache bypassing, where memory requests can selectively bypass the cache, is one solution that can help to mitigate the cache resource contention problem.

In this paper, we propose coordinated static and dynamic cache bypassing to improve application performance. At compile-time, we identify the global loads that indicate strong preferences for caching or bypassing through profiling. For the rest global loads, our dynamic cache bypassing has the flexibility to cache only a fraction of threads. In CUDA programming model, the threads are divided into work units called thread blocks. Our dynamic bypassing technique modulates the ratio of thread blocks that cache or bypass at run-time. We choose to modulate at thread block level in order to avoid the memory divergence problems. Our approach combines compile-time analysis that determines the cache or bypass preferences for global loads with run-time management that adjusts the ratio of thread blocks that cache or bypass. Our coordinated static and dynamic cache bypassing technique achieves up to 2.28X (average 1.32X) performance speedup for a variety of GPU applications.

1. Introduction

In recent years, we have witnessed the success of Graphics Processing Units (GPUs) employed for performance acceleration. The emergence of CUDA and OpenCL programming models enables easy utilization of GPUs. This has led to a corresponding proliferation of applications that are ported to GPUs. To achieve high performance, GPU programmers tend to launch a large number of threads to drive the parallel resources on GPUs. However, massive thread level parallelism does not always ensure good performance. Recent studies show that the performance of GPU applications are often limited by the memory subsystem [14, 23], especially for general purpose codes with irregular memory access patterns. Thus, the need for on-chip memory and data locality optimization for GPUs is increasingly urgent.

Initially, GPUs only employ scratchpad memory as their on-chip memory. However, scratchpad memory is highly constrained in its capabilities. To use scratchpad memory, programmers have to determine its data allocation at design or compile time. It is not appropriate for the applications with diverse access patterns, which naturally prefer cache instead of scratchpad memory. Hence, to address a wider range of workloads, recent GPU architectures have introduced caches together with scratchpad memory as the on-chip memory. For example, state-of-the-art NVIDIA Kepler architecture has both L1 data cache and scratchpad memory and allows an option for the users to configure their sizes.

In reality, however, GPU applications are often unable to effectively utilize the caches. The large number of threads in GPU applications tend to issue a large number of memory requests in a near short period. This leads to high cache contention and thus low cache hit rate. We examined all the applications in Rodinia [5] and Parboil [41] benchmark suites. For a typical 16KB L1 data cache, the L1 data cache hit rate is only about 27.1% on average. Furthermore, the massive thread level parallelism and low L1 data cache hit rate also cause serious congestion in memory requests service and other cache related resources such as miss status holding registers (MSHRs), leading to pipeline stall. Hence, in many cases GPU caches may even hurt the performance [18].

In this paper, we propose coordinated static (compile-time) and dynamic (run-time) cache bypassing to improve the GPU application performance. Cache bypassing, where the memory requests can selectively bypass the cache, is effective to mitigate the cache contention and related pipeline stall. At compile-time, we first classify the global load instructions based on their localities through profiling. For the global load instructions that indicate strong preferences (either very good or bad localities), we choose to cache or bypass them for all the threads. For the rest of global load instructions, we choose to bypass them for a fraction of threads and balance the caching and bypassing at run-time.

In CUDA programming model, threads are organized into groups called thread blocks.[1] The threads within a thread block can synchronize among themselves through barriers and share data through scratchpad memory. We choose to modulate the

[1]Thread block in CUDA is equivalent to work-group in OpenCL.

978-1-4799-8930-0/15/$31.00 ©2015 IEEE

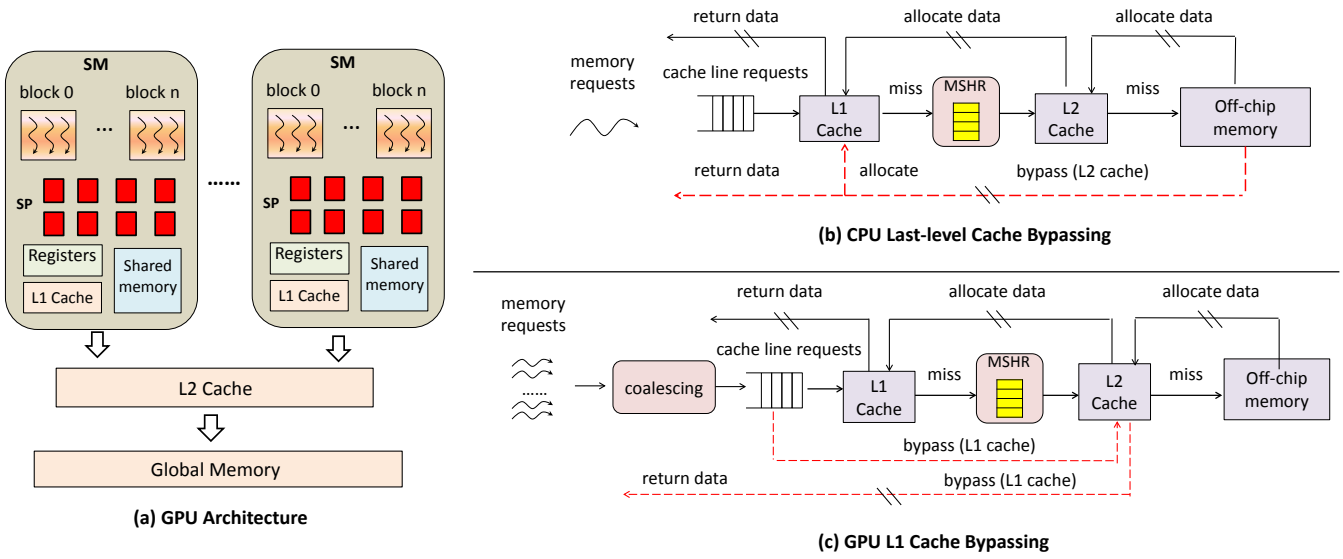

Figure 1: GPU Architecture and Cache Bypassing on CPUs and GPUs.

threads that cache or bypass at thread block level at run-time to avoid the memory divergence problem [32]. More clearly, among the set of active thread blocks (e.g., the thread blocks execute simultaneously), we choose a subset of them to bypass the cache and let the rest of active thread blocks use cache. By allowing a subset of thread blocks to use the cache, it helps to reduce the cache contention and pipeline stall as the number of memory requests to the cache is reduced. Meanwhile, the data localities for the thread blocks that access the cache can still be exploited. Note that we do not penalize the thread level parallelism in order to reduce the cache contention and pipeline stall. By doing this, we maintain massive threading for high throughput.

This paper makes the following contributions.

- We propose a coordinated static and dynamic cache bypassing optimization framework for GPUs. It employs compile-time techniques to provide global locality hints to the hardware, and employs the hardware to dynamically adjust the ratio of thread blocks that use cache.
- We propose profiling-based static analysis that classifies the global loads into three categories based on their localities and encode the classification into the load instructions at compile-time.
- We develop run-time management techniques that modulate the ratio of thread blocks that use or bypass the cache.

We evaluate our technique using a wide range of applications from Rodinia [5], Polybench [15], MapReduce [16], and Parboil [41]. Compared to always turning cache on, our coordinated bypassing achieves up to 2.28X (geometric mean is 1.32X) performance speedup for 16KB cache. Compared to the state-of-the-art bypassing techniques [46] and [19], our technique increases the performance speedup from 1.11X and 1.17X to 1.32X. We also compare our technique with the state-of-the-art two-level thread scheduling policy [34], cache-aware thread parallelism management technique [23], and cache-conscious wavefront scheduling (CCWS) technique [39]. Experiments demonstrate that our technique increases the speedup from 1.04X, 1.09X, and 1.15X to 1.32X.

The rest of the paper is organized as follows. Section 2 describes the background on GPU architecture and cache bypassing interfaces. Section 3 presents the coordinated cache bypassing framework and motivation. Section 4 and Section 5 detail the implementations of static and dynamic cache bypassing components, respectively. Section 6 discusses the experimental results. Section 7 and 8 describe the related work and conclusion.

2. Background

2.1. GPU Architecture

GPUs are composed hierarchically. A GPU is composed of multiple Streaming Multiprocessors (SM). Multiple Streaming Processors (SP), registers, scratchpad memory (a.k.a shared memory), and L1 data cache are grouped together into a SM. SMs coordinate the single-instruction multiple-data (SIMD) style execution of all of the SPs.[2] Figure 1 (a) describes our target GPU architecture.

The threads of a GPU application are first organized into groups called thread blocks. The threads in a thread block can share data through shared memory and synchronize among each other using barrier instructions. When a kernel launches, a thread block as a whole is assigned to one SM for execution. The number of threads that can simultaneously execute on one SM is limited by the available resource of an SM [1]. When all the threads in a thread block complete execution, the thread block is committed and a new thread block is dispatched to the corresponding SM.

[2]We use NVIDA and CUDA terminology in this paper, but our techniques are common to other architectures and OpenCL programming models.

Each thread block is further organized into warps, with each warp having 32 threads. The threads within a warp have to execute in a SIMD manner. Hence, the performance of the application might be affected if the threads in a warp exhibit different control or memory behavior. More clearly, if the threads in a warp take different paths at a conditional branch, the execution of the paths will be serialized, leading to resource idle and thus performance degradation [10]. Similarly, if the memory requests issued by a warp have different memory latencies, the warp has to stall until the longest memory request completes [32, 33].

2.2. Cache Bypassing

Cache bypassing has been shown to be effective in mitigating cache contention and thrashing for general purpose processors [22, 43]. Recently, GPU architectures such as NVIDIA Fermi and Kepler start to incorporate cache bypassing for the same reason. However, cache bypassing interfaces are implemented differently on the two platforms due to their distinct architecture differences.

Figure 1 (b) and (c) give the implementations of cache bypassing on modern CPUs and GPUs, respectively. On CPUs, L1 cache hit rate is often high and thus L1 cache is looked up for every memory request. Cache bypassing is mainly used for last-level cache (LLC) in recent studies [8, 11, 24, 44]. More clearly, depending on the localities of the data, cache bypassing can selectively choose to allocate it on LLC or bypass LLC. As shown in Figure 1 (b), the requested data from memory is allocated to the L1 cache only and the L2 cache is bypassed. Hence, cache bypassing on CPUs is used at the data allocation stage.

The large number of parallel threads on GPUs leads to low per-thread cache capacity and serious cache contention. Hence, the L1 cache hit rates of GPU applications are low. As a result, L1 data cache is a good candidate for bypassing. On GPUs, memory requests can choose to bypass the L1 cache as shown in Figure 1 (c). Later, the requested data is directly forwarded to the processors without filling the L1 cache. Such cache bypassing has been implemented on NVIDIA Fermi (GTX480, C2050) and Kepler (GTX680, K10) architectures. In addition, massive threading also causes congestion in cache miss handling resources such as MSHRs, cache lines, etc. For example, the current L1 cache miss can not be served if there is no MSHR entry available as shown in Figure 1 (c). Then, the pipeline has to be stalled until the requested resource is available. By forwarding some of the memory requests to L2 cache directly, cache bypassing helps to reduce the resource congestion and pipeline stall.

Cache bypassing support on the current generations of GPUs is still rudimentary. For example, on NVIDIA Fermi and Kepler, programmers can choose to use or bypass cache entirely for the program by controlling the compilation flags (*-dlcm=ca* or *-dlcm=cg*). However, such coarse-grained solutions do not exploit the entire spectrum of opportunities

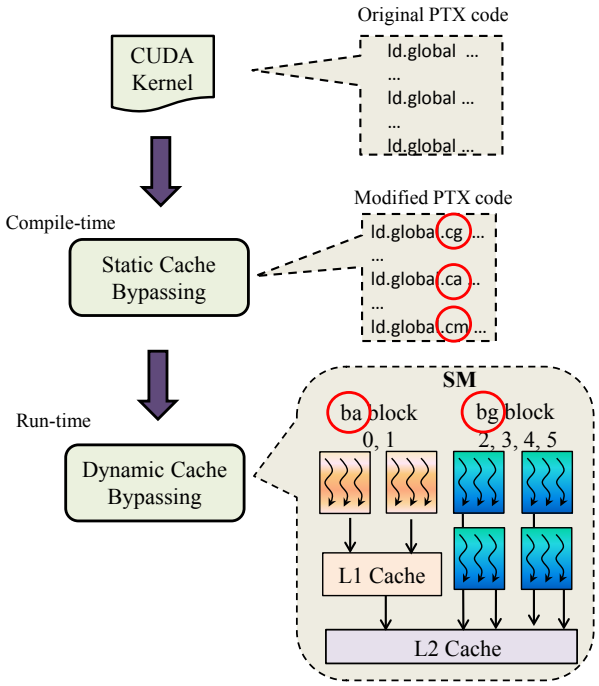

Figure 2: System Overview. Static bypassing classifies global loads into three categories using tags *ca*, *cg*, and *cm*. Thread blocks tagged with *ba* use L1 cache while thread blocks tagged with *bg* bypass L1 cache.

provided by caching and bypassing.

Currently, GPUs do not cache global store data in the L1 cache as L1 caches are not coherent for global data. Hence, global stores update L2 cache directly by invalidating the matched copies in the L1 cache. Thus, similar to previous works [18, 19, 46], we only focus on global memory loads.

3. System Overview and Motivation

The coordinated static and dynamic cache bypassing optimization framework is implemented based on GPGPU-Sim [4] (version 3.2.1) compilation and run-time system as illustrated in Figure 2. The input is the GPU application code in PTX format, which is the intermediate representation of CUDA code. The static cache bypassing component analyzes the PTX code and classifies the memory requests based on their localities and encodes the classification through instruction set extension at compile-time. At run-time, the dynamic cache bypassing component honors the cache or bypass decisions for the memory requests with strong preferences to cache or bypass, but has the flexibility to adjust the behavior for the rest of the memory requests at thread block level.

3.1. Static Cache Bypassing Overview

The memory access patterns used in general purpose codes are diverse. Some of the accesses may inherently have good localities such as *array[tid % 2]*, *array[bid]*, where the *tid* and *bid* are the thread and thread block identifer, respectively. In

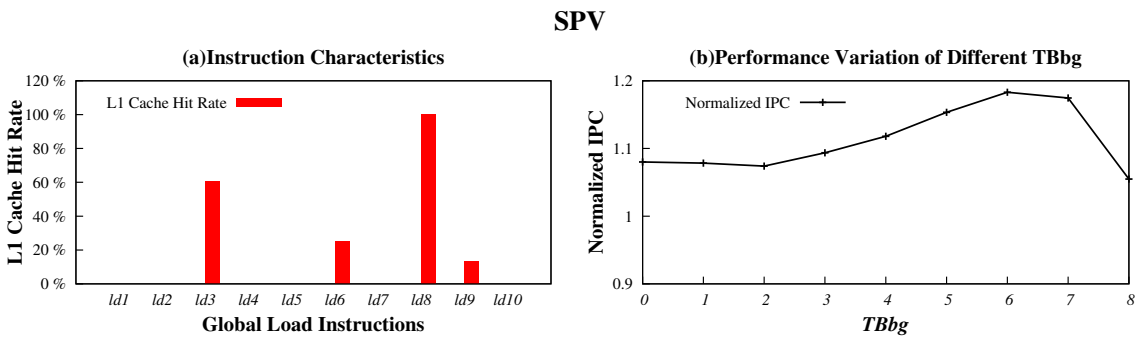

Figure 3: Motivation for Coordinated Cache Bypassing using *SPV* application.

contrast, some of the accesses may inherently have bad localities such as those used for reading and writing streaming data. Finally, there are some accesses that have neither good nor bad localities. We refer them as medium localities accesses.

Static cache bypassing component first detects all the global load instructions. On our target platform, the global load instruction in PTX is in the following format,

$$\text{ld.global.}l1\_cache\_tag.\ ...,...,$$

where $l1\_cache\_tag$ field is used to specify the cache option. Natively, there exists two options for $l1\_cache\_tag$: ***ca*** and ***cg***. If a global load is tagged with ***ca***, then all the threads of the kernel will use the cache when executing the load. If a global load is tagged with ***cg***, then all the threads of the kernel will bypass the cache when executing the load. Here, we extend the global load instructions with a third option, called ***cm***, for the loads with medium localities. Static cache bypassing component classifies all the global loads into three categories: ***ca***, ***cg***, and ***cm***, and encodes the classification into the global load instruction as shown in Figure 2. We use ***ca*** for the instructions with good locality; ***cg*** for the instructions with bad locality; ***cm*** for the instructions with medium locality. Section 4 provides the implementation details of static cache bypassing component.

3.2. Dynamic Cache Bypassing Overview

Dynamic cache bypassing component first decodes the classification for global loads encoded by static cache bypassing component. For the global loads tagged with ***ca*** or ***cg***, it honors the decision made at compile-time. However, we leave the cache or bypass decisions for the global loads tagged with ***cm*** to dynamic cache bypassing at run-time. The goal of dynamic cache bypassing is to strike a balance between caching and bypassing — let some of the threads bypass the cache to mitigate the cache contention and resource congestion, and the other threads still use the cache to exploit the data localities.

In CUDA programming model, the threads within a warp are executed in lockstep. If the threads in a warp take different latencies to fetch data (e.g., cache and bypass differently), then the warp has to be stalled until the longest memory request completes. In addition, GPU applications often use barrier instructions to synchronize the threads in the same thread block. If the threads take different time to reach the barrier due to different cache or bypass behaviors, the earlier threads have to wait for the later threads to move forward together. In both scenarios, performance will be degraded as the resources occupied by the waiting threads are idle. Thus, we choose to modulate the threads that cache or bypass at thread block level to ensure that the threads in the same thread block have the same cache or bypass behavior.

To distinguish the cache and bypass thread blocks, each thread block is assigned with a tag when it is dispatched. We use tags ***ba*** or ***bg***. If a thread block is tagged with ***ba***, then all the threads in it will use cache when executing the global loads tagged with ***cm***. If a thread block is tagged with ***bg***, then all the threads in it will bypass the cache when executing the global loads tagged with ***cm***. Once a GPU core fetches a global load, it determines its behavior based on the global load tag and current thread block tag as illustrated by Table 1.

Table 1: Instruction Behavior

Global Load Tag	Block Tag	Cache or Bypass
ca	ba	cache
	bg	cache
cg	ba	bypass
	bg	bypass
cm	ba	cache
	bg	bypass

We use TB_{max} to denote the maximum number of thread blocks that can execute simultaneously on an SM. TB_{max} is determined by the resource usage per thread block and the available resource of an SM [1]. We use TB_{ba} and TB_{bg} to denote the number of active thread blocks that are tagged with ***ba*** and ***bg*** at run-time, respectively. Based on the definition, we have $TB_{ba} + TB_{bg} = TB_{max}$. [3]. Dynamic cache bypassing component adjusts the TB_{bg} at run-time and determines the tag for each thread block. For example, in Figure 2, thread blocks 2, 3, 4, and 5 are tagged with ***bg*** and thread blocks 0 and 1 are

[3]For the last round of thread blocks execution, $TB_{ba} + TB_{bg} \leq TB_{max}$

tagged with ***ba***. Section 5 provides the implementation details of dynamic cache bypassing component.

3.3. Motivation

We illustrate the benefit of coordinated cache bypassing using application *SPV*. Figure 3 (a) shows the L1 cache hit rates for all the global loads in *SPV* through profiling. There are 10 global loads in total. Among all the loads, $ld8$ has very high cache hit rate ($> 99\%$) and thus we classify it as good locality load (tag ***ca***). However, $ld1$, $ld2$, $ld4$, $ld5$, $ld7$, and $ld10$ have very low cache hit rate ($< 1\%$) and thus we classify them as bad locality loads (tag ***cg***). For the remaining loads ($ld3$, $ld6$, $ld9$), we classify them as medium locality loads (tag ***cm***). Figure 3 (b) shows the performance by varying the TB_{bg} from 0 to 8 (Tb_{max} is 8 for *SPV*). The performance is normalized to the default setting, where all the global loads use the cache. Clearly, our coordinated cache bypassing has high potential for performance improvement. For the leftmost point in Figure 3 (b), where all the global loads use the cache except the loads tagged with ***cg*** (bad locality), it already gives about 1.1X speedup. By varying the TB_{bg}, the performance can be further accelerated to 1.2X speedup ($TB_{bg} = 6$). Note that, in this experiment, we fix the TB_{bg} throughout the kernel execution. But in our dynamic cache bypassing, we can adjust the TB_{bg} during the kernel execution at run-time.

4. Implementation of Static Bypassing

In this section, we present the implementation details of our static cache bypassing component. The goal of static cache bypassing is to classify the global loads into three categories, good, bad, and medium localities. Intuitively, the global loads with high cache hit rates have good locality (tag ***ca***), while the global loads with low cache hit rates have bad locality (tag ***cg***). The rest of the global loads have medium locality (tag ***cm***).

For a GPU application with N global loads, we use ld_i to denote the i_{th} global load in the program order. Our static bypassing component first analyzes the PTX code of GPU kernels and constructs the kernel control flow graph (CFG). Then, we extend the CFG to load control flow graph (LCFG). More clearly, in LCFG, each node represents a global load. A basic block in CFG will be splitted into multiple nodes in LCFG if it contains multiple loads, each node corresponding to one load. Then, we connect the splitted nodes with control flow edges one by one. After that, we replace a basic block with a dummy node if it does not contain any load.

Next, we annotate LCFG with some important high-level program information. First, each load is associated with an access destination, which indicates the data array it loads from. Second, parallel programs often design cooperative algorithms between threads through synchronization barriers. We insert a sync node between two nodes in LCFG if there exists synchronization between them in the source CUDA program. Finally, we add one backward edge from the last node to the first node. This is because in multi-thread GPU

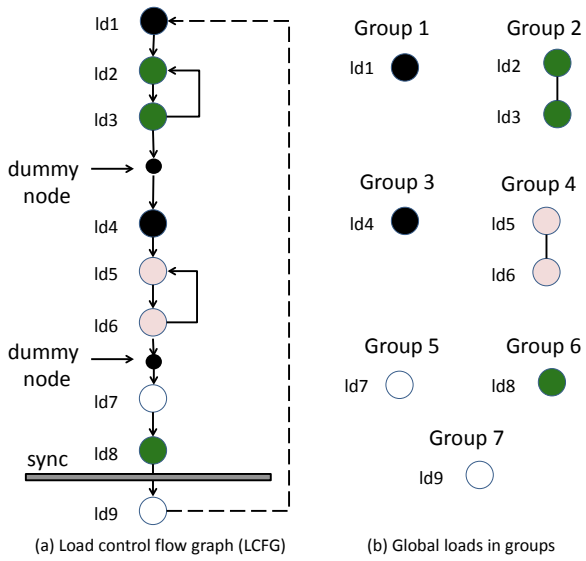

Figure 4: Annotated Load Control Flow Graph. Loads to the same array use the same color.

programs, a new thread block will be dispatched when a thread block retires. Figure 4 (a) shows an example of LCFG. In this example, LCFG contains nine loads that access four different arrays. It also contains two loops, and one synchronization barrier between ld_8 and ld_9.

The data brought into cache by one load may be later referenced by other loads. Using LCFG, we can identify the loads that have inter-instruction locality. For the applications we study, there is no aliasing between data arrays. Thus, only the loads to the same data array may have inter-instruction locality. More importantly, through empirical experiments, we find that it is unlikely for two loads to enjoy inter-instruction locality if there exist loops or synchronization in between. It is because that loops will issue a large number of loads iteratively and synchronization requests all the threads in a thread block to issue the loads before the barrier. In either case, the cache content might be flushed by the execution of a large number of loads. Hence, in our analysis, we assume that two loads have the opportunities for inter-instruction locality if they access the same data array and there exists at least one path in LCFG between them that does not have loops and synchronization.

Then, we build inter-instruction locality graph $G = (V, E)$. The node $v_i \in V$ represents ld_i. Nodes v_i and v_j are connected via an undirect edge if they have inter-instruction locality opportunities. Then, we find all the connected components in G. The connected components can be obtained using polynomial time breadth first search. Each connected component is considered as a group. Figure 4 (b) shows the connected components of the LCFG in Figure 4 (a). In this example, it divides the loads in LCFG into seven groups.

We use the following metrics to characterize the intra-instruction locality for each load and inter-instruction locality for each group (e.g. connected component in G).

- $access(i)$: the number of accesses of ld_i.

- $hit(i)$: the number of L1 cache hits of ld_i when ld_i has exclusive use of cache.
- $hit(g)$: the number of L1 cache hits of group g when the loads in group g have exclusive use of cache.

We obtain these metrics through profiling. More clearly, we collect $access(i)$ and $hit(i)$ by using ***ca*** tag for ld_i and ***cg*** tags for the rest of global loads. Similarly, we collect the $hit(g)$ by using ***ca*** tags for the loads in group g and ***cg*** tags for the rest of global loads. This profiling interface has been supported in recent GPUs such as Kepler GTX680.

Finally, we classify the global loads into three categories. The global loads classification algorithm is detailed in Algorithm 1. We first build the inter-instruction locality graph $G = (V,E)$ from LCFG. Then, for each group, we compute the extra cache hits if the inter-instruction locality of the group is exploited. Then, for each $v \in G$, we compute the potential cache hits gained by caching v (line 7). This includes the hits of v and the average extra cache hits per node. Then, we use two thresholds $HighThres$ and $LowThres$ to divide the global loads into three categories. In this paper, we empirically choose 0.7 for $HighThres$ and 0.3 for $LowThres$.

Algorithm 1: Global Loads Classification Algorithm

```
1  Build inter-instruction locality graph G(V,E) from LCFG ;
2  foreach g ∈ G do
3  |  extra(g) = hit(g) − Σ_{v∈g} hit(v)
4  end
5  foreach v ∈ V do
6  |  Let v belong to group g ;
7  |  Hit = hit(v) + extra(v)/size(g) ;
8  |  if Hit ≥ HighThres × access(v) then
9  |  |  L1_cache_tag ← ca;
10 |  else if Hit ≤ LowThres × access(v) then
11 |  |  L1_cache_tag ← cg;
12 |  else
13 |  |  L1_cache_tag ← cm;
14 |  endif
15 endfch
```

We use profile-based static analysis to characterize the localities of loads and divide them into three main categories. Overall, the profiling overhead is very small (see experiment section). In this work, we use different inputs for profiling and evaluation. As we will show in the experiments, our profile-based static analysis can accurately characterize the behavior of loads. But for the applications that have unpredictable behaviors, a more detailed profiling may be necessary. We leave it as future work.

5. Implementation of Dynamic Bypassing

Dynamic cache bypassing aims to balance the caching and bypassing at run-time by adjusting the number of active thread blocks that use cache (TB_{ba}) and the number of active thread blocks that bypass cache (TB_{bg}). To achieve this goal, it leverages online learning. The learning process consists of three steps as follows,

- *Step* 1. Set an initial value for TB_{bg} when the kernel starts execution.
- *Step* 2. Start the timer when there are TB_{bg} active thread blocks tagged with ***bg*** and keep TB_{bg} unchanged for a sampling period P.
- *Step* 3. Compare the performance metric of the current sampling period with that of histories and update TB_{bg}.

Initially, we set $TB_{bg} = TB_{max}$. Then, we iteratively execute step 2 and 3 until all the thread blocks are executed.

In Step 2, we start the timer only when the number of active thread blocks tagged with ***bg*** is the target number (TB_{bg}). We use the lifetime of a thread block as the sampling period due to the discrete nature of grid execution. This also ensures that a thread block does not change its tag during the execution and avoids the memory divergence problem due to different cache or bypass behaviors.

In Step 3, we first define the Cache Hit Stall Score ($CHSS$) as the performance metric and then use it in online learning to adjust TB_{bg}. $CHSS$ is defined as follows,

$$CHSS = \frac{Hits \cdot L2\_Latency}{Stall \cdot WarpCount} \tag{1}$$

More clearly, $CHSS$ is a function of the number of L1 cache hits ($Hits$), pipeline stall ($Stall$) caused by L1 cache congestion, and active warps ($WarpCount$). $L2\_Latency$ is a constant, which denotes the latency of L2 cache access. Cache bypassing improves performance by mitigating the cache contention and pipeline stall. $CHSS$ considers both the latency saved by cache hits ($Hits \cdot L2\_Latency$) and the latency paid for pipeline stall ($Stall \cdot WarpCount$). Ideally, we want to maximize the cache hits and minimize the stall together. Thus, we seek a TB_{bg} value to make $CHSS$ as large as possible.

In order to do that, we maintain a $CHSS$ table $CHSS[]$, where $CHSS[a]$ gives the achieved $CHSS$ value for the sampling period where $TB_{bg} = a$. At the end of Step 3, $CHSS[]$ table (entry $CHSS[TB_{bg}]$) is updated using the $CHSS$ value collected for the current period. Then, we determine the TB_{bg} for the next sampling period by comparing $CHSS[TB_{bg}]$ with its neighbors $CHSS[TB_{bg}+1]$ and $CHSS[TB_{bg}-1]$,

$$TB_{bg} = \begin{cases} TB_{bg}+1, & \text{if } CHSS[TB_{bg}+1] \text{ is the maximal} \\ TB_{bg}, & \text{if } CHSS[TB_{bg}] \text{ is the maximal} \\ TB_{bg}-1, & \text{if } CHSS[TB_{bg}-1] \text{ is the maximal} \end{cases}$$

Next, we illustrate how to determine the tags (***bg*** or ***ba***) for each thread block. When a thread block is dispatched, we first compute Cur_{bg}, the number of active thread blocks that are tagged with ***bg***. Note that Cur_{bg} may be different from the target TB_{bg} as thread blocks are dispatched and committed on the fly. Let us use $tag[bid]$ to denote the tag assigned to thread block bid (thread block identifier). We determine $tag[bid]$ by

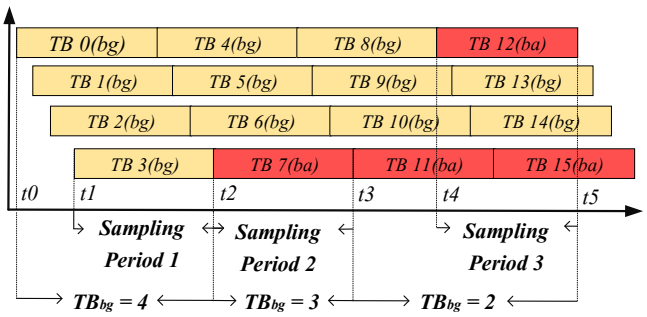

Figure 5: Sampling period, TB_{bg} update, and tag assignment.

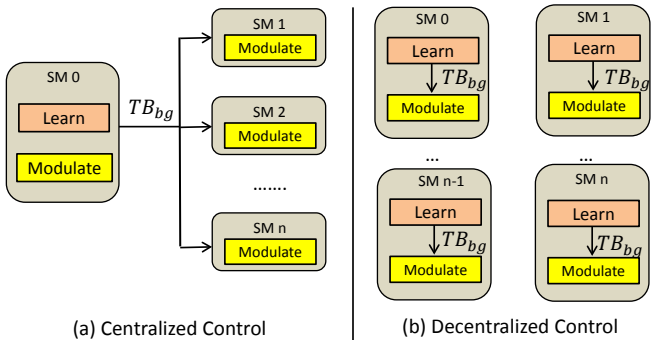

Figure 6: Centralized control vs decentralized control.

comparing the Cur_{bg} with the target TB_{bg} as follows,

$$tag[bid] = \begin{cases} \textbf{\textit{bg}}, & \text{if } Cur_{bg} < TB_{bg} \\ \textbf{\textit{ba}}, & \text{otherwise} \end{cases} \quad (2)$$

Figure 5 illustrates the sampling period, TB_{bg} update and tag assignment to each thread block using an example. In this example, we assume $TB_{max} = 4$. Initially, we set $TB_{bg} = 4$. Thus, the first four thread blocks are all tagged with ***bg***. We start the timer for the first period at $t1$ as there are exactly 4 thread blocks that are tagged with ***bg*** at $t1$. Before a sampling period (the lifetime of thread block 3) elapses, thread blocks 0, 1, 2 are committed and thread blocks 4, 5, and 6 are dispatched. When thread block 4 starts, there are three blocks tagged with ***bg*** (thread blocks 1, 2, 3). Thus, $Cur_{bg} = 3$. Then, according to Equation 2, the tag of thread block 4 is set to ***bg***. Similarly, the tags of thread blocks 5 and 6 are set to ***bg***, too. After the first sampling period, TB_{bg} is changed to 3. Then, according to Equation 2, the tag of thread block 7 is set to ***ba***. Steps 2 and 3 are repeated until all the thread blocks are executed. Note that after the second sampling period, TB_{bg} is changed to 2 and we start the timer for the next sampling period at $t4$ as there are two thread blocks (thread block 11 and 12) tagged with ***bg*** at $t4$.

Finally, we propose two different approaches with tradeoff in learning cost and flexility. One is centralized control and another one is decentralized control as shown in Figure 6. In centralized control, only SM0 performs online learning and broadcasts its decision (TB_{bg}) to the other SMs. All the SMs modulate the thread blocks in the same way. In decentralized control, each SM performs learning individually. Given a GPU application, its thread blocks are dispatched to SMs in a round-robin manner. SMs may have different workloads as thread blocks may have different computation and memory behaviors. Compared to centralized control, decentralized control allows flexible adjustment for different SMs but at expense of higher learning cost. We will compare these two approaches in the experiment section.

Hardware Implementation Overhead. Dynamic cache bypassing component requires small hardware changes. First, on GPUs, the number of simultaneously active thread blocks, TB_{max} is limited (e.g. 8). The size of $CHSS[]$ table is $TB_{max} +$ 1. Thus, the $CHSS[]$ table only requires 72 bytes. Second, for each thread block, we only need one extra bit for its tag (***ba*** or ***bg***). In fact, we only need to allocate extra TB_{max} bits per SM, where each bit represents the tag for one active thread block. Finally, two extra 4-bit registers are required: one for Cur_{bg} and another one for TB_{bg}. Cur_{bg} is updated when thread blocks tagged with bg are retired or dispatched.

6. Experiments

We implement our coordinated static and dynamic cache bypassing optimization framework based on GPGPU-sim [4] (version 3.2.1) using the configurations in Table 2. We evaluate our technique using a wide range of applications. We have examined all the applications in Parboil [41] and Rodinia [5], and some applications from PolyBench [15] and MapReduce [16]. Note that the evaluated applications are also used in the state-of-the-art GPU cache management techniques [19, 23, 34, 39, 46]. In general, applications can be categorized into two categories: cache sensitive and cache insensitive applications. For cache insensitive applications, they have much less number of memory instructions than computing instructions. The details of the applications are shown in Table 3. For each application, we simulate to the end of its execution.

Table 2: GPGPU-Sim Configuration

# Compute Units (SM)	15
SM configuration	32 cores, 700MHz
Resource per SM	Max 1536 threads, Max 8 thread blocks, 48KB shared memory, 32K registers
Scheduler	2 warp schedulers per SM, RR policy
L1 Data Cache	16/32/48KB, 128B block, LRU replacement policy, 32 MSHR entries
L2 Unified Cache	6 banks, 128KB/bank, 16-way, 128B block

Our static bypassing component characterizes the localities of loads and categorizes them into three categories, good locality (tag ***ca***), bad locality (tag ***cg***), and medium locality (tag ***cm***) based on profiling. The profiling overhead is very small for

Table 3: Application Characteristics

Cache Sensitive Applications							
Application Characteristics				Profiling Input		Evaluation Input	
Application	Suite	*abbr.*	# ca/cm/cg Loads	Input Size	# inst	Input Size	# inst
bfs	Rodinia [5]	BFS	1/7/0	65K nodes	0.9M	1M nodes	41M
backprop	Rodinia [5]	BKP	5/2/5	32768	36M	65536	72M
cfd	Rodinia [5]	CFD	1/19/21	193K	120M	0.2M	150M
convolution	Polybench [15]	CVR	0/10/0	7.9M	47M	9.4M	96M
gemm	Polybench [15]	GEM	3/0/0	512*512*256	0.7B	512*512*128	0.4B
gram	Polybench [15]	GRM	0/5/0	128*65536	0.3B	64*65536	0.2B
histogram	Parboil [41]	HIS	0/17/0	20	15M	10000	15M
hotspot	Rodinia [5]	HSC	0/0/2	1024	0.44B	512	0.11B
lud-int	Rodinia [5]	LUI	0/3/0	128	1.3M	256	5.9M
needle1	Rodinia [5]	NW1	0/0/18	4096	3.2M	8192	6.4M
needle2	Rodinia [5]	NW2	0/0/19	4096	3.2M	8192	6.5M
pvc-map	MapReduce [16]	PVM	0/7/0	270K	0.4M	1.2M	0.8M
pvc-copy	MapReduce [16]	PVC	0/3/0	270K	2.2M	1.2M	2.9M
spmv	Parboil [41]	SPV	0/3/7	2M	5.2M	60M	83M
srad2	Rodinia [5]	SRD	4/5/0	1024*2048	37M	2048*2048	74M
ssc-bit	MapReduce [16]	SSB	0/6/0	256*128	3.7M	512*128	4.6M
ssc-copy	MapReduce [16]	SSC	0/1/2	256*128	2.8M	512*128	6.2M
stencil	Parboil [41]	STE	0/0/12	128*128*32	27M	512*512*64	91M
streamcluster	Rodinia [5]	SCK	0/9/0	32768*65546	77M	65536*65536	0.15B
3d-convolution	Polybench [15]	3DC	5/5/1	1024*512*512	20M	512*512*512	20M
Cache Insensitive Applications							
cutcp	Parboil [41]	CUT	0/2/0	405K	0.15B	6.5M	0.6B
gaussian	Rodinia [5]	GAU	1/3/0	matrix4	6.2K	matrix16	6.4K
lbm	Parboil [41]	LBM	0/0/20	1*4.1M	0.56B	1000*4.1M	0.56B
mri-q	Parboil [41]	MRQ	0/0/5	32*32*32	0.61B	64*64*64	48B
pathfinder	Rodinia [5]	PAT	0/0/3	20000*100*20	26M	100000*100*20	0.13B
sad	Parboil [41]	SAD	1/0/0	50K	5.6M	4M	0.45B
sgemm	Parboil [41]	SGM	0/0/18	225K	0.1M	15M	4.9B
srad1	Rodinia [5]	SRA	1/0/0	256*128	7.6M	520*458	14M
tpacf	Parboil [41]	TPA	0/6/0	10391	14B	487	4.1B

all the applications. It ranges from a few milliseconds to one second. We evaluate our technique using different profiling and evaluation inputs. The inputs used in the experiments are described in Table 3. As shown, different problem sizes are used for profiling and evaluation. Table 3 also gives the total number of global loads and the number of global loads in each category for every application. Clearly, different applications have different distributions of global loads. For example, application *GEM* has good localities throughout the program. Thus, all of its loads are tagged with ***ca***. In contrast, *HSC* consistently prefers bypassing and all loads are tagged with ***cg***. For application *CFD*, its loads exhibit diverse behaviors and are divided into multiple categories.

In the following, we perform five sets of experiments to evaluate our coordinated cache bypassing technique. In Section 5, we propose two dynamic cache bypassing mechanisms. We use the centralized control as the default mechanism in the experiments. First, we demonstrate the performance benefit of coordinated cache bypassing for 16KB data cache. Second, we compare coordinated cache bypassing with the state-of-the-art bypassing techniques [19, 46], two-level thread scheduling policy [34], cache-aware thread parallelism management technique [23], and cache-conscious wavefront scheduling (CCWS) technique [39]. Third, we evaluate coordinated cache bypassing using different cache sizes. Fourth, we compare two different dynamic bypassing mechanisms. Finally, we show how the TB_{bg} is modulated over time.

6.1. Performance Results

Figure 7 shows the IPC improvement of static cache bypassing alone, dynamic cache bypassing alone, and coordinated cache bypassing for cache sensitive applications for 16KB cache. The baseline is the default setting, where all the global loads use the L1 data cache. In static cache bypassing alone, all the loads tagged with ***cm*** and ***ca*** use the L1 data cache, while the loads tagged with ***cg*** bypass the L1 data cache; the dynamic component is disabled. In dynamic cache bypassing alone, the ratio of thread blocks that use or bypass cache are adjusted

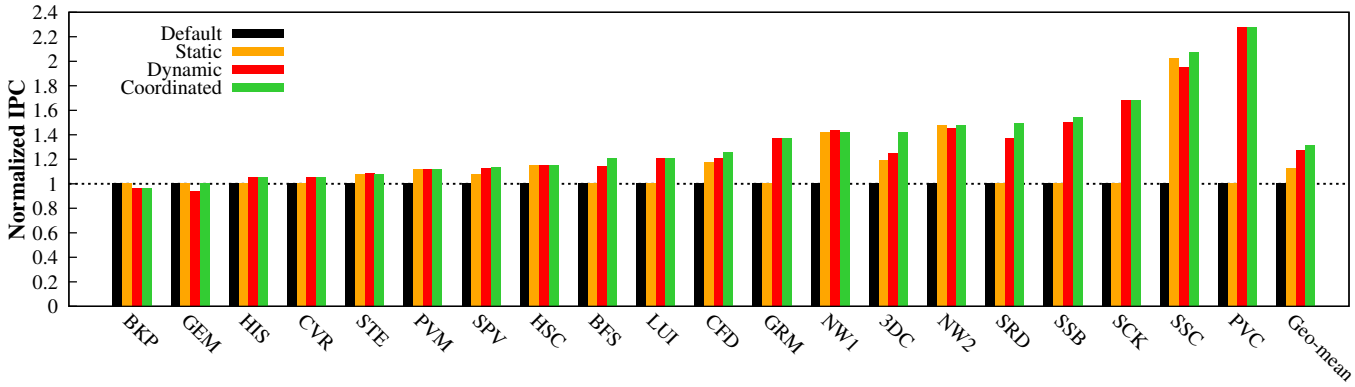

Figure 7: Performance Results for Cache Sensitive Applications on 16KB Cache.

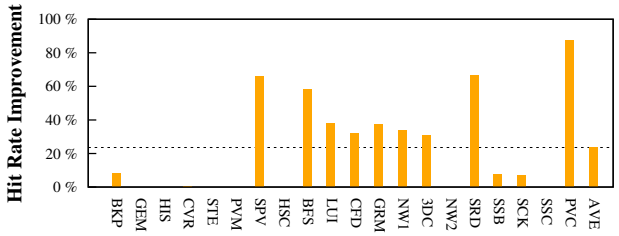

Figure 8: L1 Cache Hit Rate Improvement (16KB).

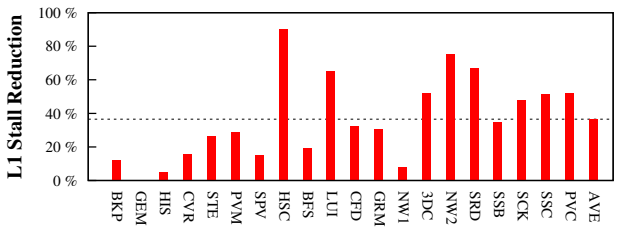

Figure 9: Reduction on Pipeline Stall Caused by L1 Cache Congestion (16KB).

at run-time for all the loads. In other words, all the loads are considered as medium locality loads (tag ***cm***). Static alone, dynamic alone, and coordinated bypassing achieve average 1.13X, 1.28X, and 1.32X performance speedup, respectively.

Static bypassing alone performs well for the applications that only have loads tagged with *cg*. For example, all loads in *HSC*, *NW1*, *NW2*, and *STE* are tagged with *cg*. Bypassing these bad locality loads, static bypassing achieves better performance than the default setting. Applications that have *cm* loads desire fine-gained and adaptive dynamic bypassing. For example, *PVC* only contains *cm* loads and it does not benefit from static bypassing, but dynamic bypassing effectively boosts its performance to 2.28X. We notice that dynamic bypassing may slightly degrade the performance (3.1% for *BKP*, 5.6% for *GEM*) due to the learning cost. For these two applications, the initial value used for TB_{bg} (TB_{max}) is not optimal, the learning process takes some time to adjust TB_{bg}. In general, coordinated cache bypassing combines the benefits of static and dynamic bypassing. It performs consistently well for all of the applications.

Coordinated cache bypassing improves performance by mitigating the cache contention, pipeline stall, or a combination of the two. Here, we support this argument with quantitative experiments. Figure 8 shows the cache hit rate improvement and Figure 9 shows the pipeline stall reduction due to the L1 cache congestion for 16KB cache, respectively. Cache hit rate improvement is calculated as $hit\_after - hit\_before$, where $hit\_after(hit\_before)$ is the cache hit rate after (before) coordinated cache bypassing. Overall, coordinated cache bypassing improves the cache hit rate by 23.6% and reduces the pipeline stall by 36.5%.

For applications *HIS*, *CVR*, *STE*, *PVM*, *HSC*, *NW2*, and *SSC*, cache bypassing can not improve their cache hit rates as they inherently have poor localities. The performance improvements are attributed to the pipeline stall reduction. These applications have high number of L1 cache misses and every cache miss requests an entry in the MSHR table as shown in Figure 1. If there is no free entry in the MSHR table, the memory stage is stalled. Thus, high number of L1 cache misses subsequently leads to serious MSHR congestion and thus pipeline stall. Cache bypassing can effectively alleviate the MSHR congestion and pipeline stall by forwarding the memory requests to L2 cache directly.

For applications *SPV*, *BFS*, *LUI*, *CFD*, *SRD*, and *PVC*, their loads have good or medium localities. However, without cache bypassing, they suffer from low L1 cache hit rate due to cache contention. Coordinated cache bypassing reduces cache contention by bypassing the bad locality loads and adjusting the number of thread blocks that use the L1 data cache. Thus, for these applications, L1 cache hit rates are effectively improved as shown in Figure 8. As L1 cache hit rates improve, the number of L1 cache misses are reduced. This may subsequently alleviate the MSHR congestion. For application *PVC*, its high performance speedup is attributed to both cache hit improvement and pipeline stall reduction.

We also evaluate coordinated bypassing for the cache insensitive applications in Table 3. Coordinated cache bypassing only improves the performance by 0.94% for 16KB cache on average. This is expected as in general cache optimization techniques have little impact on them.

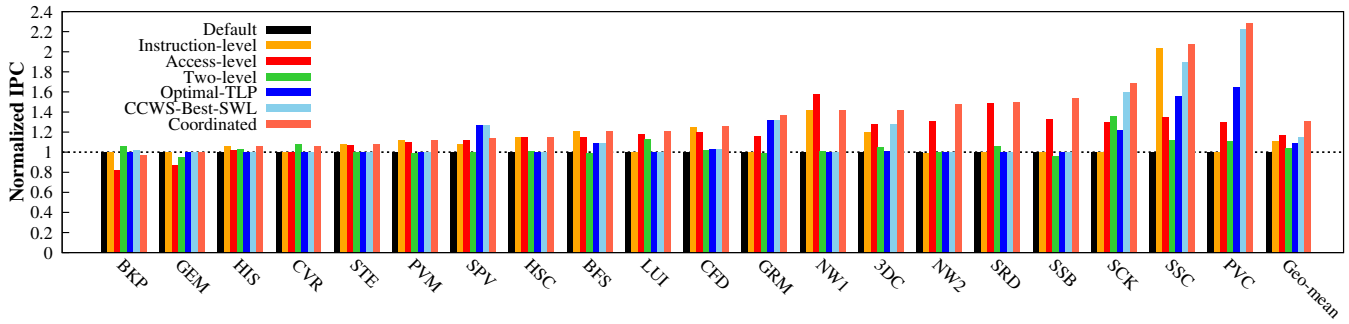

Figure 10: Performance Comparison with State-of-the-art Optimization Techniques.

6.2. Comparison to the State-of-the-art Techniques

In this subsection, we compare our technique with the state-of-the-art GPU cache management techniques.

Instruction-level Bypassing. Instruction-level bypassing techniques which determine the cache or bypass behavior for each global load instruction at compile-time are proposed in [18, 46]. We compare with the heuristic algorithm in [46]. It is shown that the heuristic algorithm gives comparable performance to the optimal algorithm.

Access-level Bypassing. Access-level bypassing triggers bypassing when cache associativity stall is encountered [19]. In [19], request reordering is used together with cache bypassing. However, request reordering requires extra hardware resource (a few KB SRAM). Thus, for a fair comparison, we only compare with the cache bypassing component.

Two-level Scheduler. A two-level thread scheduler is described in [34]. In their technique, the ready warps are divided into different groups. Then, the scheduler prioritizes the groups by scheduling the warps only from one group until all the warps in it are stalled.

Optimal-TLP. It is shown that the maximal thread level parallelism (TLP) may not lead to the best performance due to the cache contention problem [23]. By allocating fewer thread blocks, cache contention is reduced and thus performance can be improved. We compare with the Optimal-TLP solution in [23], which is obtained via exhaustively testing all the possible thread block allocations.

CCWS-Best-SWL. Cache-conscious warp scheduler (CCWS) is proposed to improve the intra-block cache locality [39]. In [39], two schedulers are proposed: a static optimal wavefront limiting (SWL) scheduler and a victim cache-based scheduler. We choose to compare with SWL scheduler as it gives better performance. We use greedy-then-oldest (GTO) policy in SWL as shown in [39].

The baseline scheduler is the round-robin scheduler. We use it with our *Coordinated Bypassing*, *Instruction-level Bypassing*, *Access-level Bypassing*, and *Optimal-TLP*. For *Two-level Scheduler* and *CCWS-Best-SWL*, we use the scheduling policies specified above.

Figure 10 shows the normalized IPC for all the techniques. The baseline setting is the default setting, where all the global loads use cache. On average, the performance speedups of *Instruction-level Bypassing*, *Access-level Bypassing*, *Two-level scheduler*, *Optimal-TLP*, *CCWS-Best-SWL*, and *Coordinated Bypassing* are 1.11X, 1.17X, 1.04X 1.09X, 1.15X, and 1.32X, respectively.

Discussion. *Instruction-level* bypassing is a coarse-grained technique. It mainly benefits applications that consist of global loads tagged with ***ca*** and ***cg***. However, for the applications with ***cm*** loads such as *PVC*, *SRD*, and *SCK*, there is little performance improvement. *Coordinated* approach improves instruction-level bypassing as it identifies medium locality loads and allows a fraction of threads to use cache for them. *Access-level* bypassing is a fine-grained technique. However, it lacks of a global view of locality. Such locality oblivious approach may lead to poor bypassing decision. For example, *Access-level* bypassing gives negative performance improvement for *BKP* and *GEM* as it chooses to bypass the loads with good localities. In contrast, *Coordinated* approach classifies the loads through compile-time analysis. This guarantees that it will not bypass the loads with good localities.

Two-level and *CCWS-Best-SWL* aim to improve intra-block access locality by either prioritizing the threads or limiting the number of active warps. *Optimal-TLP* reduces the cache contention at the expense of less number of thread blocks. However, all of them do not analyze the localities of loads. Thus, they may fail to improve the performance for the applications with poor localities. For example, all the three approaches give no gain for application *HSC*. In contrast, coordinated bypassing relies on static analysis to filter out the poor localities loads (tagged with *cg*) and this helps to reduce the cache contention and related pipeline stall. More importantly, limiting the number of active warps (*CCWS-Best-SWL*) or thread blocks (*Optimal-TLP*) may lead to low resource utilization and thread level parallelism. For example, for *SSC*, *Optimal-TLP* approach achieves 1.56X speedup by allocating only one thread block. However, the maximum number of allowed simultaneous thread blocks for *SSC* is 8. Thus, only 1/8 of the computing resources are utilized in *Optimal-TLP*. On average, the ratio of allocated thread blocks to maximum thread blocks for *Optimal-TLP* is 66%, which implies low computing resources utilization. In contrast, *Coordinated* approach does not penalize the thread level parallelism. Using

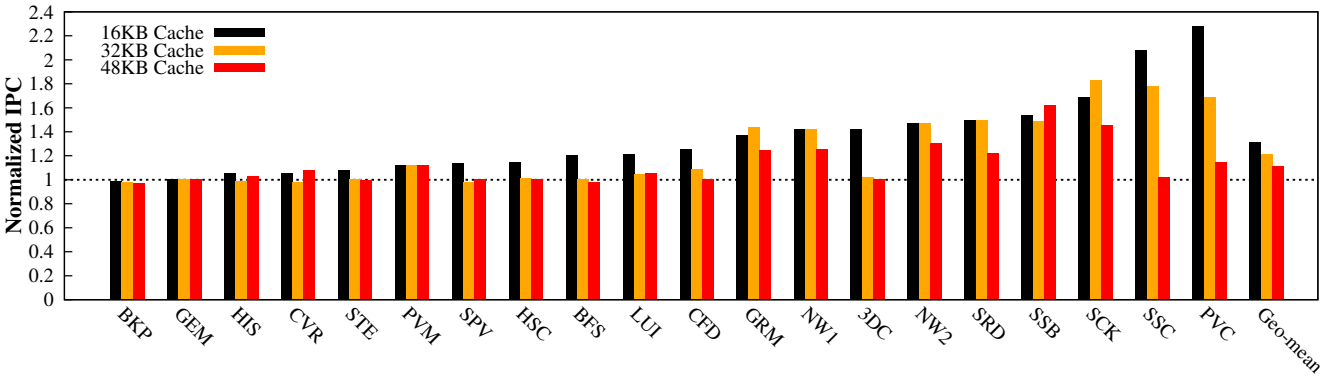

Figure 11: Performance Results for Different Size of Caches.

cache for a fraction of threads and let the rest bypass cache, coordinated bypassing not only reduces the cache contention, but also maintains the high thread level parallelism of GPU applications. For application *SSC*, *Coordinated* approach gives 2.08X speedup. Finally, as indicated in [23, 39], *Optimal-TLP* and *CCWS-Best-SWL* are used as performance improvement upper bounds. In reality, they are impractical solutions as they rely on exhaustive evaluation.

6.3. Sensitivity to Different Size of Caches

Figure 11 shows the performance speedup of coordinated bypassing for different size of caches. It achieves average 1.32X, 1.21X, and 1.11X speedup for 16KB, 32KB, and 48KB caches, respectively. We observe that the speedup over the baseline setting diminishes as the cache size increases. This is because a larger cache leads to larger per-thread cache capacity and less cache contention. For example, for applications *HSC* and *3DC*, their working set can fit into 32KB and 48KB caches, and thus our coordinate bypassing does not give any speedup for them.

However, large caches do not solve all the problems. For example, applications (*PVM* and *SSB*) have intrinsic poor temporal localities and larger caches do not help to improve the cache hit rate and performance. But cache bypassing is useful for them as it can help to reduce the pipeline stall. Furthermore, large caches not only increases the access latency and power consumption, but also implies less area for other hardware components given the area limitation. For example, NVIDIA Fermi and Kepler architectures use unified cache and shared memory design. In this unified design, cache size increase means shared memory size decrease. However, this may hurt the performance as there might be fewer active thread blocks executing concurrently when the shared memory size decreases. In contrast, cache bypassing improves the performance by maintaining the shared memory capacity and the thread level parallelism.

6.4. Centralized vs Decentralized Control

In Section 5, we propose two dynamic cache bypassing mechanisms. One is centralized control and another one is decentralized control. Centralized control has low learning cost while decentralized control provides flexibility for different SMs. We compare these two mechanisms on a typical 16KB L1 data cache for all the cache sensitive applications. On average, centralized control performs slightly better than decentralized control (3.2%) due to its lower learning cost.

6.5. Thread Block Modulation

Figure 12 shows how TB_{bg} is modulated over time in coordinated bypassing with two examples. Initially, TB_{bg} is set to TB_{max}. During run-time, TB_{bg} is dynamically adjusted.

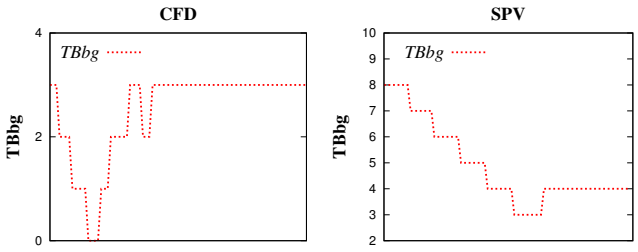

Figure 12: TB_{bg} Modulation

7. Related Work

GPUs have been widely adopted to accelerate general-purposed applications. However, fully utilizing the performance potential of GPUs is not a trivial task. Significant hurdle lies in the performance tuning. The state-of-the-art of GPU performance modeling and optimization techniques include performance modeling [3, 17, 35], control flow divergence optimizations [7, 9, 32, 37, 38], resource utilization improvement through multitasking [2, 29, 36], data layout transformations [30, 42], and on-chip memory designs [13, 14, 28].

Among all the optimization techniques, memory system optimizations are increasingly important as more and more general purpose applications with diverse and irregular memory access patterns are ported to GPUs. Thread scheduling policies are optimized in [12, 20, 34] to preserve the temporal locality. Thread level parallelism management techniques are

discussed in [23, 39, 40]. They alleviate memory pressure by thread/warp throttling. Memory scheduling policy [33] and data prefetching [21, 27] are also studied in the literature.

Cache bypassing that aims to selectively bypass memory requests is an effective technique to mitigate the cache contention and cache related resource congestion. Both static [45] and dynamic approaches [22, 25, 26, 31, 43, 44] are proposed for general purpose processors. The techniques for CPUs mainly use cache hit rate as the guidance performance model for cache bypassing. However, on GPUs, cache hit rate based models do not always predict the performance well due to the distinct architecture features including massive parallelism, resource congestion, and memory divergence [6, 18, 33].

A few recent studies have explored cache bypassing for GPUs. Instruction-level cache bypassing techniques that analyze the data access patterns of loads and estimate the memory traffic at compile-time are proposed in [18, 46]. However, their techniques only allow global loads to either use cache for all the threads or bypass cache for all the threads. In contrast, our coordinated bypassing introduces medium locality loads and has the flexibility to use cache for a fraction of threads. A dynamic approach that leverages on requests reordering and access-level bypassing is described in [19]. Requests reordering buffers the recent accesses and allows rearrangement in service sequence. Cache bypassing is triggered by cache associativity stall. However, such cache bypassing ignores data locality and may bypass the data with potential good localities. In contrast, our coordinated cache bypassing has a global view on data locality and this ensures that it will not bypass the data with good locality. Moreover, our coordinated cache bypassing can work together with the request reordering technique in [19]. Recently, Chen et al. [6] propose an adaptive cache management technique by combining the protect distance-based cache bypassing [8] and thread throttling [39] to improve the cache performance. Similar to our findings, they also demonstrate that cache bypassing can effectively mitigate cache pressure with less penalty on thread level parallelism compared to the pure thread throttling technique [39]. In contrast, our coordinated cache bypassing alleviates cache contention by statically identifying and bypassing the bad localities loads and dynamically adjust the thread blocks that use the cache for medium localities loads. Compared to [6], our coordinated cache bypassing can further improve the thread level parallelism and thus performance without thread throttling. Finally, their cache hit rate based cache bypassing and thread throttling technique requires complex hardware extension while our technique can be implemented with very small hardware change.

8. Conclusion

GPUs are increasingly important for performance acceleration due to their tremendous computing power. Recent GPUs have adopted caches to improve the memory performance for general purpose applications with irregular memory access patterns. However, GPU applications often fail to benefit from caches due to the cache contention and resource congestion caused by the massive thread parallelism.

In this paper, we propose a coordinated static and dynamic cache bypassing optimization framework. At compile-time, coordinated cache bypassing classifies the global loads into three categories through profiling and encodes the classification into instructions. The classification provides global locality hints to the hardware. At run-time, coordinated cache bypassing dynamically adjusts the ratio of thread blocks that use cache to reduce cache contention and pipeline stall. The experiments demonstrate that coordinated cache bypassing achieves substantial performance improvement for a variety of GPU applications. Compared to the state-of-the-art instruction-level and access-level cache bypassing techniques, our technique increases the performance speedup from 1.11X and 1.17X to 1.32X. Compared to the state-of-the-art two-level thread scheduling policy, cache-aware thread parallelism management technique, and cache-aware thread scheduling technique, coordinated cache bypassing increases the performance speedup from 1.04X, 1.09X, and 1.15X to 1.32X.

Acknowledgments: This work was partially supported by the National Science Foundation China (No. 61300005, 61373026, and 61261160501), National High Technology Research and Development Program of China (No. 2012AA010903), 973 project (No. 2013CB329000), Tsinghua University Initiative Scientific Research Program, and Huawei. We thank the anonymous reviewers for their feedback.

References

[1] "NVIDIA." occupancy Calculator. http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html.

[2] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte, "The case for GPGPU spatial multitasking," in *IEEE 18th International Symposium on High Performance Computer Architecture*, ser. HPCA, Feb 2012, pp. 1–12.

[3] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-M. W. Hwu, "An adaptive performance modeling tool for gpu architectures," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10, 2010, pp. 105–114.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS, April 2009, pp. 163–174.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, ser. IISWC, 2009, pp. 44–54.

[6] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. W. Hwu, "Adaptive cache bypass and insertion for many-core accelerators," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'47, 2014.

[7] Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An accurate GPU performance model for effective control flow divergence optimization," in *2012 IEEE 26th International Parallel Distributed Processing Symposium*, ser. IPDPS, May 2012, pp. 83–94.

[8] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'45, 2012, pp. 389–400.

[9] W. W. L. Fung and T. M. Aamodt, "Thread block compaction for efficient SIMT control flow," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA, 2011, pp. 25–36.

[10] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'40, 2007, pp. 407–420.

[11] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and insertion algorithms for exclusive last-level caches," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA, 2011, pp. 81–92.

[12] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA, 2011, pp. 235–246.

[13] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'44, 2011, pp. 465–476.

[14] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'45, 2012, pp. 96–106.

[15] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Innovative Parallel Computing*, ser. InPar, 2012.

[16] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT, 2008, pp. 260–269.

[17] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA, 2009, pp. 152–163.

[18] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS, 2012, pp. 15–24.

[19] W. Jia, K. A. Shaw, and M. Martonosi, "MRPB: Memory request prioritization for massively parallel processors," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, ser. HPCA, 2014.

[20] A. Jog, O. Kayıran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2013, pp. 395–406.

[21] A. Jog, O. Kayıran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA, 2013, pp. 332–343.

[22] T. L. Johnson and W.-M. W. Hwu, "Run-time adaptive cache hierarchy management via reference analysis," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA, 1997, pp. 315–326.

[23] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT, 2013, pp. 157–166.

[24] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'43, 2010, pp. 175–186.

[25] M. Kharbutli and D. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, pp. 433–447, April 2008.

[26] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA, 2001, pp. 144–154.

[27] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for GPGPU applications," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'43, Dec 2010, pp. 213–224.

[28] Y. Liang, Z. Cui, K. Rupnow, and D. Chen, "Register and thread structure optimization for GPUs," in *Proceedings of the 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013, pp. 461–466.

[29] Y. Liang, H. Huynh, K. Rupnow, R. Goh, and D. Chen, "Efficient GPU spatial-temporal multitasking," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, 2014.

[30] Y. Liang, Z. Cui, S. Zhao, K. Rupnow, Y. Zhang, J. L. Douglas, and D. Chen, "Real-time implementation and performance optimization of 3D sound localization on GPUs," in *Design, Automation & Test in Europe Conferenc & Exhibition*, ser. DATE.

[31] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'41, 2008, pp. 222–233.

[32] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA, 2010, pp. 235–246.

[33] S. Mu, Y. Deng, Y. Chen, H. Li, J. Pan, W. Zhang, and Z. Wang, "Orchestrating cache management and memory scheduling for GPGPU applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 8, pp. 1803–1814, Aug 2014.

[34] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'44, 2011, pp. 308–317.

[35] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A detailed GPU cache model based on reuse distance theory," in *IEEE 20th International Symposium on High Performance Computer Architecture*, ser. HPCA, Feb 2014, pp. 37–48.

[36] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2013, pp. 407–418.

[37] Y. Park, J. J. K. Park, H. Park, and S. Mahlke, "Libra: Tailoring SIMD execution using heterogeneous hardware and dynamic configurability," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'45, 2012, pp. 84–95.

[38] M. Rhu and M. Erez, "Maximizing SIMD resource utilization in GPGPUs with SIMD lane permutation," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA, 2013, pp. 356–367.

[39] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'45, 2012, pp. 72–83.

[40] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'46, 2013, pp. 99–110.

[41] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "The IMPACT Research Group, Parboil Benchmark Suite." [Online]. Available: http://impact.crhc.illinois.edu/Parboil/parboil.aspx

[42] I.-J. Sung, G. Liu, and W.-M. W. Hwu, "DL: A data layout transformation system for heterogeneous computing," in *Innovative Parallel Computing*, ser. InPar, May 2012, pp. 1–11.

[43] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, ser. MICRO'28, 1995, pp. 93–103.

[44] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr., and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO'44, 2011, pp. 430–441.

[45] Y. Wu, R. Rakvic, L.-L. Chen, C.-C. Miao, G. Chrysos, and J. Fang, "Compiler managed micro-cache bypassing for high performance EPIC processors," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO'35, 2002, pp. 134–145.

[46] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD, 2013, pp. 516–523.