Adaptive and Transparent Cache Bypassing for GPUs

Ang Li[*,†], Gert-Jan van den Braak[*], Akash Kumar[‡], and Henk Corporaal[*]

[*]Eindhoven University of Technology, Eindhoven, The Netherlands
[†]National University of Singapore, Singapore
[‡]Technische Universität Dresden, Dresden, Germany
{ang.li, g.j.w.v.d.braak}@tue.nl, akash.kumar@tu-dresden.de, h.corporaal@tue.nl

ABSTRACT

In the last decade, GPUs have emerged to be widely adopted for general-purpose applications. To capture on-chip locality for these applications, modern GPUs have integrated multi-level cache hierarchy, in an attempt to reduce the amount and latency of the massive and sometimes irregular memory accesses. However, inferior performance is frequently attained due to serious congestion in the caches results from the huge amount of concurrent threads. In this paper, we propose a novel compile-time framework for adaptive and transparent cache bypassing on GPUs. It uses a simple yet effective approach to control the bypass degree to match the size of applications' runtime footprints. We validate the design on seven GPU platforms that cover all existing GPU generations using 16 applications from widely used GPU benchmarks. Experiments show that our design can significantly mitigate the negative impact due to small cache sizes and improve the overall performance. We analyze the performance across different platforms and applications. We also propose some optimization guidelines on how to efficiently use the GPU caches.

CCS Concepts

•Computer systems organization → Multiple instruction, multiple data; •Software and its engineering → Source code generation;

Keywords

Cache bypassing; GPUs; Thread throttling

1. INTRODUCTION

Graphics Processing Units (GPUs), the coprocessor originally designed predominantly for graphic rendering, nowadays has been proven unexpectedly successful in the domain of general-purpose applications (GPGPU) [1, 2, 3]. A crucial issue that confines the peak performance delivery, however, is the vast and sometimes irregular memory accesses from massively concurrent threads. This enforces considerable pressure on the bandwidth and efficiency of the memory system [4]. To reduce memory traffic and latency, modern GPUs have widely adopted hardware-managed cache hierarchies [5, 6]. However, traditional cache management strategies are mostly designed for CPUs and sequential programs; replicating them directly on GPUs may not deliver expected performance, as GPUs' relatively smaller caches can be easily congested by thousands of threads, causing serious contention and thrashing. Table 1 lists the L1 cache[1] capacity, thread volume and per-thread L1 cache share for the state-of-the-art multithreaded processors. As can be seen, the per-thread cache share for GPUs is much smaller than for CPUs, which indicates that the useful data fetched by one thread is very likely to be evicted by other threads before actual (re-)usage. Such thrashing condition destroys locality and impairs performance. Moreover, the excessive incoming memory requests, particularly in an accessing burst period (e.g. the starting and ending phases of a kernel) if concerning the SIMT execution model [7] (see Section 2.1), can lead to significant delay when threads are queuing for the limited resources in caches, e.g. miss buffers, MSHR entries, a certain cache set, etc. [8, 9].

Table 1: Threads vs Caches.

Processor	L1 Cache	Thd/Core	Cache/Thd
AMD Warsaw	16 KB	1	16 KB
Intel Haswell	32 KB	2	16 KB
Intel Xeon-Phi	32 KB	4	8 KB
Oracle M5	16 KB	8	2 KB
Nvidia Fermi	48 KB	1536	32 B
Nvidia Kepler	48 KB	2048	24 B
Nvidia Maxwell	24 KB	2048	16 B
AMD Radeon-7	16 KB	2560	6.4 B

A naive response is to extend the cache capacity. However, it sacrifices the valuable die area that may otherwise be dedicated for more computation facilities. Therefore, instead of prototyping "big-cached" GPUs, designers are more prone to throttle the thread volume in order to reach a good balance between multithreading degree and cache efficiency [10, 11].

Traditional thread throttling mechanisms either advise

[1]In this paper, L1 cache refers to L1 data cache only.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SC '15, November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-3723-6/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2807591.2807606

users to refine their code using an ideal multithreading degree predicted from parsing the source code [12, 13], or suggest hardware modifications in the thread scheduler to limit active thread count, so as to match access footprints with the cache capacity [11, 14, 15]. However, the thread number from the user part is often determined by the underlying algorithm; altering it is not straightforward and may lead to the reformation of the algorithm, which demands tremendous user efforts. On the other hand, restricting threads according to cache capacity in the scheduler may diminish the utilization of the computation units and off-chip memory bandwidth [16]. Besides, the smart scheduler often requires either a brilliant compile-time analyzer or a powerful runtime detector. Further, the orchestrated hardware modifications can only be implemented in future products; it cannot benefit existing platforms anyway. Both of the above approaches are costly, from either application or hardware perspectives.

Thus the challenge is, *can we design a throttling mechanism that is transparent to the users and the hardware, but is still adaptive and efficient*? In this paper, we give a solution: *during compilation, we can add a threshold so that only a limited number of threads can access the cache.*

This paper makes the following contributions:

- We propose a novel and simple compile-time framework to do adaptive and transparent cache bypassing for global memory read in all three types of GPU caches: *L1*, *L2* and *read-only* caches (Section 4.2).
- We propose a static and a dynamic approach to acquire the ideal bypass threshold (Section 4.4).
- We evaluate the bypassing framework on seven GPU platforms that covers all GPU generations with general caches inside: *Fermi*, *Kepler* and *Maxwell* with compute capability 2.0 to 5.2 (Section 5).
- We propose two software methods (Section 6.1) and investigate a hardware implementation (Section 6.2) to reduce the overhead of cache bypassing.
- Finally, we propose several optimization guidelines on the utilization of GPU caches (Section 5.3).

2. BACKGROUND

In this section, we first briefly introduce the execution model of GPUs and explain why the granularity for the proposed bypassing framework should be a *warp*. We then describe the three different datapaths for *global memory read operations* which are the main target of this paper.

2.1 GPU Execution Model

Evolved from SIMD, the execution model of GPUs is named *single-instruction-multiple-threads* or *SIMT* [7, 17]. A kernel, which is a function that runs in the GPU part, includes thousands of threads that are primarily grouped into multiple *thread blocks* (TBs, also known as cooperative thread arrays (CTAs)). When a kernel launches, its TBs are dispatched to several streaming multiprocessors (SMs)[2]. Threads inside a TB are further organized as a number of execution groups that perform the same operations on different data in a lockstep manner. Such execution groups are called *warps*. In an SM, a warp is the basic unit in terms of scheduling, executing and accessing memory. If threads in a warp diverge at a point (e.g. upon *if-else*), all the branches will be executed alternatively and sequentially, with threads not belonging to the present branch being masked off, until divergent threads consolidate at a convergent point and continue the lockstep execution. If a warp is obstructed by a long latency operation, an off-chip global memory read for example, the warp scheduler will switch-in another ready warp instantly with no cost [17]. How to establish an orchestrated scheduling for good overlapping, especially considering the positive/negative impact on the memory system, recently becomes a hot research topic [14, 15, 18, 19].

[2]Although we focus on Nvidia GPUs and use CUDA terminology in the paper, the concepts also apply to AMD GPUs.

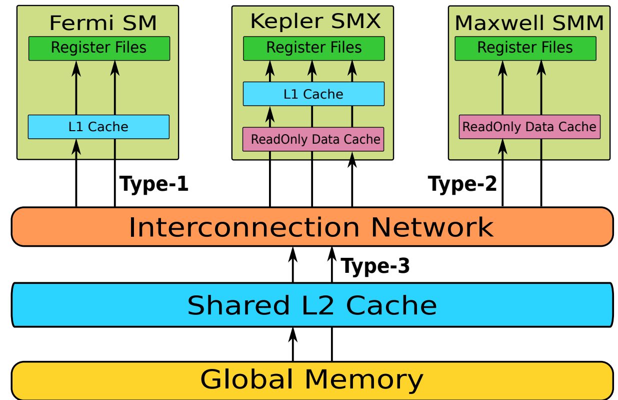

Figure 1: Global Memory Read Datapaths

2.2 GPU Memory Access Datapath

As shown in Figure 1, the GPU memory system contains registers, L1 cache, read-only data cache (via texture pipeline), interconnection network, L2 cache and off-chip global memory. Registers are private to threads. The L1 and read-only caches are shared by all resident TBs in an SM. SMs are connected to a unified L2 cache by an interconnection network. The L2 cache is generally partitioned into several banks, each of them being a buffer for a particular GDDR memory channel.

As GPUs have thousands of concurrent threads, to conserve the limited memory bandwidth and improve efficiency, simultaneous memory requests from threads in the same warp are usually combined as a group request for a cache-line sized chunk before accessing L1 cache, provided there is spatial locality across the warp. Such coalesced memory accessing pattern is often viewed as the primary step towards harvesting the performance of GPUs [20]. The L1 cache shares the same on-chip storage with the *shared memory* of an SM. Their relative sizes are reconfigurable (16/48 or 48/16 KB in Fermi and 16/48, 32/32 or 48/16 KB in Kepler). The L1 cache line is 128B. It caches both global memory read and local memory access (read and write) and is non-coherent. The local memory is generally utilized for register spilling, function calls and automatic variables [17]. Comparatively, the L2 cache is much larger with, however, a smaller cache line size of 32B. The L2 cache serves all types of memory accesses (i.e. constant access, texture access, etc) and is coherent with CPU memory.

Since the majority of memory accesses are from/to global

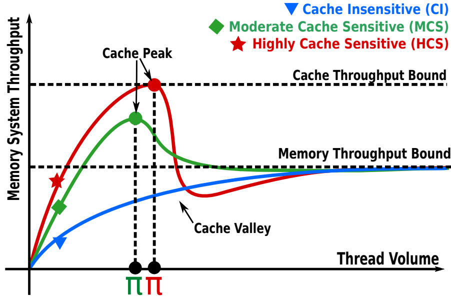

Figure 2: Plots for three types of GPU applications using the valley model.

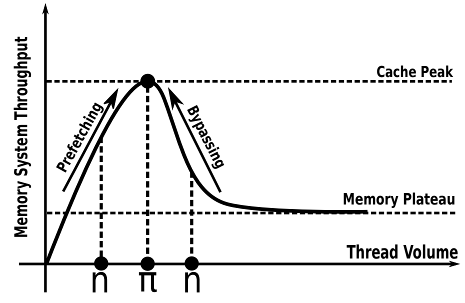

Figure 3: Climbing the cache peak from the front face via prefetching and from the back face via bypassing.

memory, the machine performance is much more sensitive to memory load than store (because load is often in the critical path as computation has dependence on the loaded data which is not the case for store). Therefore, in this paper we focus on *global memory read operations* only. Regarding such operations, from Fermi to Kepler to Maxwell, there are three different datapaths with cache involved (see Figure 1):

- **L1 datapath** (Type-1 in Figure 1): from interconnection network to register files via L1 cache in both Fermi and Kepler[3] GPUs.
- **Read-only datapath** (Type-2): from interconnection network to register files via read-only cache in Kepler[4] and Maxwell GPUs.
- **L2 datapath** (Type-3): from global memory (GDDR) to interconnection network via L2 cache in Fermi, Kepler and Maxwell GPUs.

Accordingly, there are three possible approaches for cache bypassing during global memory read: *L1 cache bypassing*, *read-only cache bypassing* and *L2 cache bypassing*.

3. VALLEY MODEL

In this section, we use a visual analytic model to intuitively describe why cache bypassing can be effective for improving GPU performance.

We first characterize all GPU applications into three categories: *cache insensitive (CI)*, *moderate cache sensitive (MCS)* and *highly cache sensitive (HCS)* [14, 22]. In [23], Guz et. al. proposed a visual analytic model to address the interaction between thread volume and shared cache for a multithreaded-manycore (MT-MC) machine. We use a refined version of their model (labeled as *valley model*) to show the variation of memory hierarchy throughput with respect to the thread volume accessing the memory. Figure 2 illustrates the general curves for the three application categories based on the valley model:

- **Cache insensitive** (CI) applications (**blue curve**) exhibit little data locality for global memory access. As thread volume expands, a higher utilization of the memory bandwidth is expected because the memory latency is increasingly hidden by context-switching among the extra threads. The memory hierarchy throughput curve increases monotonically with thread count until it approaches the bandwidth bound (denoted as *memory plateau* in Figure 3).
- **Moderate cache sensitive** (MCS) applications (**green curve**) contain moderate data locality. As thread volume increases, more cache storage is leveraged. Meanwhile, the cache hit rate also goes up. However, when the aggregated working set exceeds cache capacity, thrashing occurs, which leads to a throughput degradation. The performance rising and dropping forms a peak (denoted as *cache peak*). Since the per-thread cache share for GPUs is much smaller than CPUs (see Table 1), the GPU cache peak is more to the left in the figure, implying that it is more easily congested. With further increased threads, the cache effect becomes obscure and throughput remains consistent on the memory plateau. The thread volume that shows the best cache performance is the ideal thread volume, labeled as π.
- **Highly cache sensitive** (HCS) applications (**red curve**) the cache is even more crucial for performance. Due to ample data locality, the cache hit rate function demonstrates a super-linear behavior with increased thread count. However, beyond the cache peak, the effect of cache thrashing is also more prominent than MCS applications. This explains why beyond the cache peak, a performance valley exists (denoted as *cache valley*).

We use the MCS curve as a general case (the shape is confirmed by [14] and validated in Section 4.3) to describe why cache bypassing can benefit performance for cache sensitive applications (MCS+HCS). As shown in Figure 3, in order to attain the best performance, the thread volume (n) has to be pushed towards the ideal thread volume (π). We label this tuning process as **climbing the cache peak**. As discussed, tuning thread volume is difficult from the user part and hardware part. To develop a transparent design that *operates at compile time*, there are two strategies:

- *Cache Prefetching:* If the thread-level-parallelism is insufficient to fully exploit the memory hierarchy, we can add extra memory prefetching requests to saturate the cache, which corresponds to *climbing cache peak from the front face* (Figure 3).

[3] Only a fraction of Kepler GPUs support the L1 cache mode such as Tesla K40, K80, etc. [21].
[4] Only Kepler GPUs with compute capability larger or equal to 3.5 have the read-only cache.

- *Cache Bypassing:* If there are too many memory requests that congest the cache, some of them can be bypassed from the cache, which corresponds to *climbing cache peak from the back face* (Figure 3).

In this paper, we focus on cache bypassing. One can refer to [24, 25] and other references for GPU cache prefetching.

4. CACHE BYPASSING

The proposed adaptive bypassing designs are presented in this section: we first describe the cache operators provided by the hardware. We then propose the horizontal bypassing design and compare it with the conventional vertical design. After that, we provide a case study. Finally, we show how to acquire the ideal bypass degree via a static and a dynamic approach.

4.1 Cache Operators

Nvidia *PTX* ISA [26] introduces per-access cache operators for global memory read:

```
ld.global{.cop}{.nc}     %reg, [addr];
```

"*ld.global*" stands for global memory read. "*reg*" is the target register. "[*addr*]" is the source memory address. "*.cop*" is the cache operator which has different configurations:

- ***.ca***: cache at both L1 (if available) and L2 with default LRU replacement policy.
- ***.cg***: bypass L1 and cache at L2 with default LRU replacement policy.
- ***.cs***: *streaming* cache at both L1 (if available) and L2. It assumes that the fetched data will be accessed only once so that **evict-first** replacement policy is adopted. This option is chosen to prevent the streaming data from polluting the useful cache lines.
- ***.va***: cache as volatile. For global memory read, it is the same as ***.cs***.

In addition, "*.nc*" has two options:

- Without ***.nc***: normal memory load.
- With ***.nc***: load from L2 to register via read-only cache.

Therefore, for a specific global memory read access, we can set up the following combinations for cache bypassing corresponding to Type-1,2,3 global memory read datapaths shown in Figure 1:

- For **L1** cached access, it is *ld.global.ca*; for L1 bypassed access, it is *ld.global.cg*.
- For **read-only** cached access, it is *ld.global.nc*; for read-only bypassed access, it is *ld.global.cg*.
- For **L2** cached access, it is *ld.global.cg*; for L2 bypassed access, since there is no particular L2 bypassing operator offered while the *.cs* option that adopts eviction-first policy reduces the impact on the original cache content, due to recent data accesses, to the smallest extent, we use *ld.global.cs* as an "imperfect substitution" for L2 bypassing if there is no L1 cache. Even with L1 available, streaming-style load at both L1 and L2 is the type of load that is the closest to L2 bypassing.

```
// ============ Bypass Header ============
mov.u32       %r0, %tid.x; //Thread index
shr.u32       %r0, %r0, 5; //Warp index
setp.lt.s32   %p0, %r0, $pi$; //Set Threshold
...
// ============== L1 Cache ==============
@%p0  ld.global.ca.s32  %r9, [%rd6]; //Cache
@!%p0 ld.global.cg.s32  %r9, [%rd6]; //Bypass
...
// =========== Read-only Cache ===========
@%p0  ld.global.nc.s32  %r9, [%rd6]; //Cache
@!%p0 ld.global.cg.s32  %r9, [%rd6]; //Bypass
...
// ============== L2 Cache ==============
@%p0  ld.global.cg.s32  %r9, [%rd6]; //Cache
@!%p0 ld.global.cs.s32  %r9, [%rd6]; //Bypass
```

Listing 1: Adaptive cache bypassing

4.2 Horizontal Cache Bypassing

With the three configurations as a preamble, we can set up the horizontal cache bypassing framework. We define a **bypassing threshold**: then *for warps with index less than the threshold, they perform cached read; for warps with index larger or equal to the threshold, they do cache bypassing.*

The design is shown in Listing 1. We first use the thread index to locate the warp it belongs to (by dividing index with the warp size 32). Here, it should be noted that the PTX predefined identifier *%warpid* [26] cannot be leveraged because it returns the physical warp-slot index, not the one defined in the user-program context. Since the physical warp-slot is dynamically bound to the warps, using it may destroy intra-warp locality, which is the major resource for potential data-reuse in HCS applications [14]. Note, it is also possible to embed PTX into the CUDA program using intrinsic functions. However, working at PTX level is easier for parsing and is transparent to the users.

Depending on whether the warp index is less than the bypassing threshold π, a predicate register $p0$ is configured. Then all the global loads in the PTX program are converted to conditional accesses: *if p0 is true, cache; otherwise, bypass.* Listing 1 shows the conditional statements for the three types of GPU caches. We use warp rather than thread here as the granularity for conditional bypassing to avoid the expensive warp divergence overhead (see Section 3.1) and conserve coalesced accessing patterns (see Section 3.2).

Such a design is quite clear yet efficient: overall, only a 1-bit predicate register is required per thread as the **space cost**. The general register used for calculating warp index is only required inside the bypassing header block (see Listing 1). Since the header block is always placed at the beginning of a kernel, this register can be recycled immediately after usage. Regarding the **time cost**, except one shift operation and one predicate register setting, the major overhead is the instruction issuing delay for the one additional load (two load instructions are issued, but only one is executed). Although such overhead becomes noticeable (see Section 4.3) when there are large amounts of memory accesses, it could be reduced by merging them together since the decision for bypassing or not is constant throughout the warps' lifetime. We discuss how to reduce this overhead in Section 6.

There are three reasons for cache bypassing to be beneficial to performance: first, it mitigates cache congestion

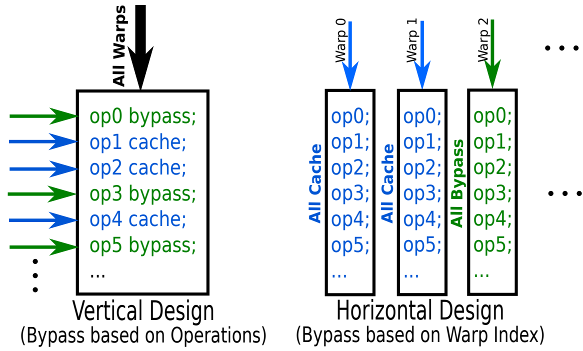

Figure 4: Bypass design approaches: vertical vs. horizontal.

so that the thread volume can match the cache capacity. In this way, the warps to be cached do not have to worry about their useful data being evicted before usage. Since the cache space per warp is sufficient to cover the accessing footprints, inner-thread and inner-warp locality are preserved and captured. Second, while the remaining warps bypass the cache, they do not need to wait for the shared resource in the cache (e.g. MSHR entry, an associative set entry, etc.) to be available before entering the memory pipeline. Last but not the least, the parallelism for the computation system is not sacrificed as we maintain the number of dispatched threads in the machine.

We would like to compare our proposed bypass design (marked as *horizontal approach*) with the existing cache operator based schemes (such as [10, 27], denoted as *vertical approach*):

- The **vertical approach** follows the conventional CPU's design paradigm that operates within a single thread scope. As shown in Figure 4, all threads/warps execute the same instruction stream while inside the stream, for each global memory read, one has to decide whether to bypass or not. The design spectrum is along the vertical *instruction direction*. Since every read instruction fetches different data, if there are m read, the design complexity is $O(2^m)$, for which m can be very large. Such a broad design space is quite difficult to traverse. Moreover, as all threads follow the same execution path, they tend to access the cache at the same time, which is more likely to congest the cache. However, this vertical design does not incur any extra time/space overhead at runtime. If assisted by a smart scheduler, it can distinguish and abolish data with little locality thus avoiding detrimental cache pollution.

- The **horizontal approach** on the other hand focuses on the most prominent characteristic of GPUs — multithreading. As shown in Figure 4, for each different warp, one has to decide if it belongs to the bypass group or cached group. However, as soon as the decision is made, all the global memory read in that warp follow. The design spectrum is along the horizontal *warp direction*. As warps in a TB are identical, the design complexity for n warps is $O(n)$, where n is less than or equal to 32. (This is true for all existing Nvidia GPUs [17]). In fact, for all applications we tested in Table 3 and all benchmarks in Rodinia [28], $n \leq 16$. Still, the memory requests may come in a burst, but bypassing enforces the number of warps that access the cache, which significantly mitigates the pressure on the cache. The drawbacks, however, are the small time and space cost.

There is no clear conclusion on which approach is better. They are **orthogonal** to each other: one focuses on code property and one focuses on concurrency. The horizontal design sees the kernel code as a blackbox, therefore, cannot distinguish those loads with little reuse. Caching such loads can be detrimental even with horizontal bypassing adopted. So a more attractive approach is a hybrid design: first bypass loads with little locality via vertical approach; then apply horizontal bypassing on the remaining loads if cache thrashing remains. We set this as a future work.

4.3 BFS Case Study

To make a clear explanation about how cache bypassing can benefit performance, a detailed case study is provided. We focus on Breadth-First-Search (BFS) in Table 3. The testing platform is Fermi (*Platform-1* in Table 2). To avoid possible interference due to insufficient data size, we use the largest dataset (*graph-1MW_6.txt*) in the benchmark. Except inserting the bypassing header and converting global memory read in the PTX routine (as in Listing 1), we do not make any other modifications to the kernel code or kernel configurations (i.e. threadgrid, threadblock, shared memory allocation, etc.). We vary the threshold value from 0 to the number of warps defined in the application (16 in this example). Also, the results for bypass-all (denoted as `bpa`) and cache-all (denoted as `cha`) are shown for reference. All result figures are the average value for 5 execution runs.

Figures 5, 6 and 7 illustrate the kernel execution time with respect to the increased bypassing threshold on L1, L2 and L1-L2 together with 16KB L1. Figures 8, 9 and 10 show the time with 48KB L1. There are two L2 bypassing results with different L1 configurations. The reason is that the L2 bypassing does not actually bypass L2 but accesses the L1 and L2 in a streaming fashion on Fermi (see Section 4.1). That's why the L1 configuration affects L2 bypassing performance. Besides, Figures 7 and 10 show the L1-L2 combining bypass effects. Comparing the six figures, we have the following observations:

First, the shapes of the curves confirm the valley model described in Section 3.1. As can be seen, π marks the position of the **cache peak**. In Figure 5, $\pi = 3$ indicates that the footprint for one warp is slightly more than 5KB (16KB/3) which is confirmed by $\pi = 9$ (48KB/9) in Figure 8. Meanwhile, the **cache valley** is quite obvious in Figure 5, as the performance degrades significantly beyond the cache peak, to a degree that is even much worse than no caching at all. A larger L1 alleviates the valley effect (from Figure 5 to Figure 8), but still, no clear gain is attained (`bpa` and `cha` are similar in Figure 8). As a comparison, for both cases bypassing filters out the excessive requests which leads to a more efficient utilization of the L1 cache.

Second, regarding L2 (Figures 6 and 9), `cha` performing better than `bpa` implies that the valley effect mitigates in L2. Also, the fact that the bypassing benefit is larger for L2 than L1 implies that the overall machine performance is more sensitive to L2 cache than L1. However, it should be noted that the best bypassing performance is always attained on L1 cache (compared with Figures 5 and 8). This means **bypassing on L2 only is not sufficient**.

Third, we also evaluate bypassing on both L1 and L2 at

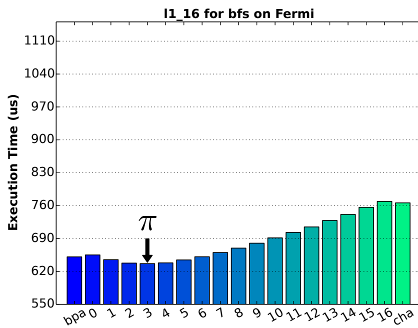

Figure 5: BFS cache bypassing on 16KB L1.

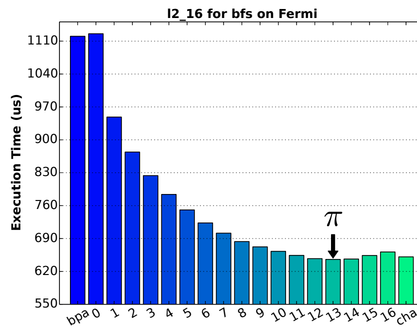

Figure 6: BFS cache bypassing on L2 with 16KB L1.

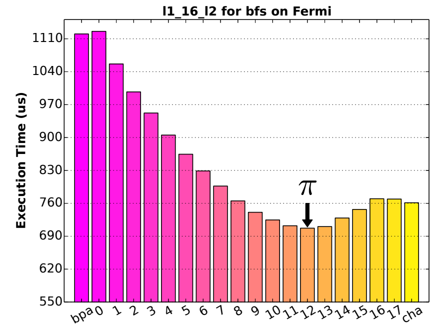

Figure 7: BFS cache bypassing on 16KB L1 and L2 simultaneously.

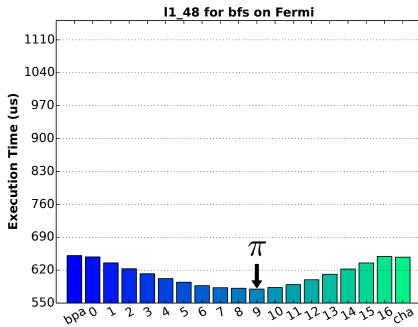

Figure 8: BFS cache bypassing on 48KB L1.

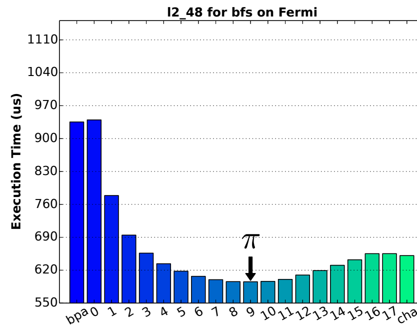

Figure 9: BFS cache bypassing on L2 with 48KB L1.

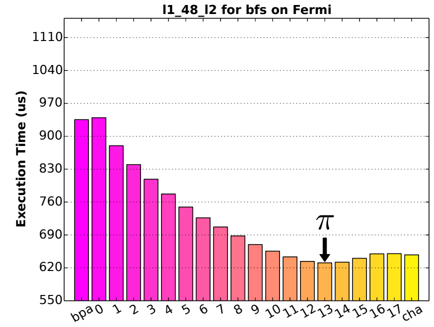

Figure 10: BFS cache bypassing on 48KB L1 and L2 simultaneously.

the same time (Figure 7 and 10). This approach is equivalent as *if cache, then cache at both L1 and L2; otherwise, bypass them all.* Note, unless using additional thresholds for L1 and L2 respectively, this is the only combining approach. As can be seen, the performance is worse than bypassing on L1 and L2 alone, which means the **bypassing benefit on L1 and L2 are not cumulative**.

Finally, about the execution overhead for bypassing. Recall that the decision boundary for caching or bypassing is "*less than*", the threshold value equals to zero thus has the same context meaning as `bpa`, but additionally contains the space and time overhead of the bypassing framework. Therefore, the small discrepancies between `bpa` and $\pi = 0$, `cha` and $\pi = 16$ in the figures are such overhead. However, it should be noted that in Figure 8, the overhead appears to be "negative" ($\pi = 0$ is less than `bpa`), this is because in the added bypassing operations (and bypassing head) may alter the original warp scheduling decision at runtime, which leads to such "rare" effect.

4.4 Acquire Ideal Bypassing Threshold

There is one question left: *how to acquire the ideal threshold* π *?* In this paper, we propose a static and a dynamic approach.

4.4.1 Static Approach

The static approach is straightforward: *just exhaustively assess all the selective values for the threshold.* Here, it highlights the advantages of horizontal bypassing over the vertical one: we only need to test 32 times at most. In fact, to reach acceptable SM occupancy, most applications have less than 16 warps in their thread block configurations. As discussed, this is true for all the applications in Rodinia and the ones we tested in Table 3. As a comparison, with only 10 loads in the kernel, a vertical scheme would have 1024 different configurations (see Section 4.2).

The advantage of the static approach is that it always returns the optimal threshold for the current dataset. Meanwhile, as GPUs normally run fast, executing a kernel 16 times is a not significant overhead. This makes the static approach a good option for program auto-tuning. The drawback, however, is that the attained threshold may correlate with the testing dataset. To overcome this "over-fitting" problem, people could use a more representative dataset or profile with multiple datasets to confirm the trend (see Section 5.2 and the supplementary file).

4.4.2 Dynamic Approach

The dynamic approach is a runtime voting method. As shown in Figure 11, we assume that there are 1024 TBs in total for the kernel and each TB has six warps based on the application logic. The kernel is then amended to generate the sampling procedure in three steps: first, seven TBs (instead of 1024) are initiated with consecutive bypass values, from $x = 0$ to $x = 6$. Then, for each TB, a thread (e.g. *tid=0*) is enforced to measure the execution time of the entire TB with the associated threshold level. The timing result is submitted **atomically** to a global-scope bypassing threshold π. Finally, if the eventual value of π equals to zero or six, the runtime manager discards the conditional state-

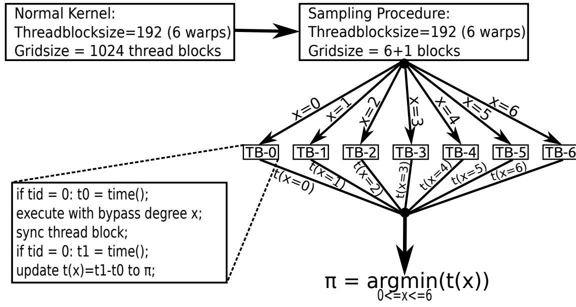

Figure 11: Sampling and voting for optimal bypassing threshold π.

ment and uses `bpa` or `cha` instead. Again, with $max(\pi) \leq 32$, we can assess all selective options with a few sampling TBs. The sampling procedure can be integrated into the runtime library to avoid user involvement.

This approach is practical and easy to implement. However, it has its drawbacks: first, it works only for L1 cache bypassing. Second, it cannot handle inter-TB unbalancing (i.e. irregular applications may have different workload for different TBs). Third and most importantly, during the sampling phase only one TB is allocated per SM, so this TB essentially occupies the entire L1 cache. But in a real execution, this is not the case; generally multiple TBs are sharing the L1 cache simultaneously. Therefore, the sampled threshold may not be accurate. Regarding this problem, as we cannot alter the TB scheduling policy via software approaches, a possible solution would be (Note, this is motivated by the latest SM-Centric programming [29]): allocate sufficient TBs to saturate all SMs. Instead of profiling different π with different TBs (as in Figure 11), we now profile in different SMs: before setting the timer, the pilot thread first acquires the sm_id of the resident SM from the special register `%smid`. Then, with different sm_id, a different π is assessed. In this way, the sampling phase simulates the actual execution more accurately.

5. EVALUATION

In this section, we validate the proposed bypassing framework. In order to evaluate the general effectiveness of the framework, we use seven GPU platforms that covers **ALL** existing Nvidia GPU generations with general cache integrated, say from compute capability (**CC**) 2.0 to 5.2[5], as shown in Table 2. We take 16 cache sensitive (HCS+MCS) applications from the Rodinia [28], Parboil [30], Mars [31] and Polybench [32] benchmarks. Since all the applications in the Mars benchmark share the common Map-Reduce kernel library, we only use one application (*SSC*). Besides, the Mars applications cannot compile properly on other platforms, so we only show the results of *SSC* for Fermi with CC-2.0. We use Normalized IPC as the performance metric since cache hit rate does not necessarily lead to better overall performance for GPUs [14, 33]. The normalized IPC here is simply the reciprocal of the execution time; we do not count the added bypass instructions when calculating IPC. Again, except inserting the bypassing header and converting global memory read in the PTX routine (as in Listing 1), we do not make other modifications to the kernel code or

[5]CC-3.2 and 5.3 are for embedded systems only.

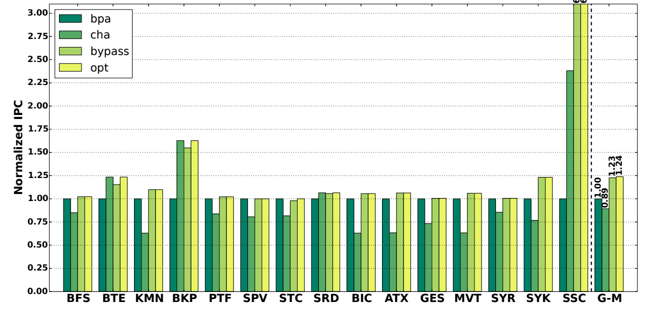

Figure 12: 16KB L1 cache bypassing on Fermi GPU.

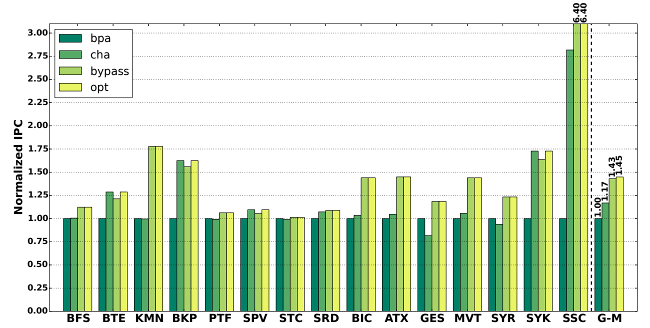

Figure 13: 48KB L1 cache bypassing on Fermi GPU.

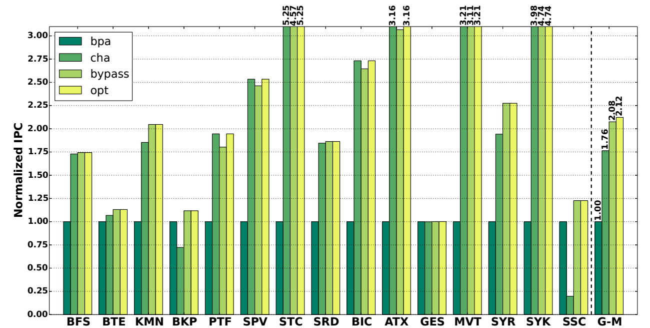

Figure 14: L2 cache bypassing on Fermi GPU.

kernel configurations. Note, for read-only caches, we only apply bypassing to loads that are accessing the "read-only" variables or arrays as the read-only caches are non-coherent. In this paper, due to page limitation, we only show the results for Platform 1 to 3. For other results, please refer to the supplementary document.

Platform-1 – Fermi: The results for 16KB L1, 48KB L1 and L2 on Fermi with CC-2.0 are shown in Figures 12, 13 and 14. For comparison purposes, we normalize the performance to `bpa`[6]. `G-M` is the geometric-mean-value. Similar to the case study in Section 4.3, the differences between *bypass* and *opt* imply the bypassing overhead.

As can be seen in Figure 12, the 16KB L1 cache is far from sufficient to cover the data footprints, which leads to the in-

[6]`bpa` is the default behavior for L1 and read-only caches of Kepler and Maxwell GPUs. However, on Fermi L1 and all L2 caches, the default is `cha`.

Table 2: Experiment Platforms

Plat.	GPU	Arch-Code	CC.	Cores	GPU Freq	Mem Band	Dri./Rtm.	CPU	gcc
1	GTX570	Fermi-110	**2.0**	15 SMx32	1464 MHz	152 GB/s	6.5/4.0	Intel Q8300	4.4.7
2	Tesla K80	Kepler-210	**3.7**	13 SMXx192	824 MHz	240 GB/s	7.0/7.0	Intel E5-2690	4.4.7
3	GTX750Ti	Maxwell-107	**5.0**	5 SMMx128	1137 MHz	86.4 GB/s	6.5/6.5	Intel i7-4770	4.4.7
4	GTX460	Fermi-104	**2.1**	7 SMx32	1400 MHz	88 GB/s	6.5/6.5	Intel i7-920	4.6.3
5	GTX690	Kepler-104	**3.0**	8 SMXx192	1020 MHz	192 GB/s	7.0/6.5	Intel i7-5930K	4.8.4
6	Tesla K40	Kepler-110	**3.5**	15 SMXx192	876 MHz	288 GB/s	6.0/6.0	Intel E5-2620	4.4.7
7	GTX980	Maxwell-204	**5.2**	16 SMMx128	1216 MHz	224 GB/s	6.5/6.5	Intel i3-4160	4.8.2

Table 3: Benchmark Characteristics

Application	Description	abbr.	Warps	Input dataset	Source
bfs	Breadth First Search	BFS	16	graph1MW_6.txt	Rodinia[28]
backprop	Back Propagation	BKP	8	65536	Rodinia[28]
b+tree	B+ Tree Operation	BTE	8	mil.txt-command.txt	Rodinia[28]
kmeans	K-means Clustering	KMN	8	kdd_cup	Rodinia[28]
stencil	3-D Stencil	STE	4	128x128x32.bin-128-128-32-100	Parboil[30]
particlefilter	Particle Filter	PTF	16	128x128x10, np:1000	Rodinia[28]
spmv	Sparse Matrix-Vector Multiplication	SPV	6	Dubcova3.mtx - vector.bin	Parboil[30]
streamcluster	Stream Cluster	STC	16	10-20-256-65536-65536-1000	Rodinia[28]
srad	Speckle Reducing Anisotropic Diffusion	SRD	16	100-0.5-502-458	Rodinia[28]
bicg	BiCGStab Linear Solver	BIC	8	default	Polybench[32]
atax	Matrix Transpose Vector Multiply	ATX	8	default	Polybench[32]
gesummv	Scalar Vector Matrix Multiply	GES	8	default	Polybench[32]
mvt	Matrix Vector Product Transpose	MVT	8	default	Polybench[32]
syrk	Symmetric Rank-K Operations	SYR	8	default	Polybench[32]
syr2k	Symmetric Rank-2K Operations	SYK	8	default	Polybench[32]
similarityscore	Similarity Measure between Documents	SSC	16	256-128	Mars[31]

ferior performance of `cha` compared with `bpa` (11% worse). Therefore, using the L1 cache naively is detrimental. However, this situation is effectively improved by the proposed bypassing scheme, which leads to 24% speedup over `bpa` and 39% over `cha`. The serious thrashing problem of 16KB L1 has been significantly mitigated by extending the cache size to 48KB. As shown in Figure 13, `cha` is 17% better than `bpa` now. Nonetheless, the effect of cache bypassing is more prominent: it demonstrates 45% speedup over `bpa` and 24% over `cha`. Regarding L2 in Figure 14, the fact that `cha` is much better than `bpa` indicates that caching in a streaming fashion (in both L1 and L2) is much worse than caching normally in L2 for most cases (except BKP and SSC). Also, our scheme achieves 1.12x speedup over `bpa` and 20% over `cha` in L2 cache. Besides, it should be noted that for all the three tests on Fermi with CC-2.0, the overhead introduced by the bypassing framework is quite small (1%, 2% and 4%).

Platform-2 – Kepler: Next we validate cache bypassing on a Kepler platform with CC-3.7 – the latest Tesla-K80 GPU. The results for 16KB, 32KB, 48KB L1, read-only and L2 caches are shown in Figure 15, 16, 17, 18 and 19, respectively.

Unlike Fermi, the L1 cache in Kepler is harmful in all configurations albeit the degree is declining (24%, 20% and 10% worse for 16KB, 32KB and 48KB). Meanwhile, the effectiveness of cache bypassing also remains evident, with a speedup of 8%, 9%, 16% over `bpa` and 42%, 36%, 29% over `cha`. The scenario for read-only cache is, however, completely different. As shown in Figure 18, the benefit of exploiting the read-only cache is 2.03x speedup of `cha` over bpa. In addition, the bypassing framework leads to 2.16x speedup over the default `bpa` approach. The condition of L2 is similar to Fermi.

Platform-3 – Maxwell: Lastly, we run the experiments on the Maxwell architecture with CC-5.0. Since Maxwell completely discards L1 cache and uses the entire on-chip storage for shared memory, we can only establish read-only

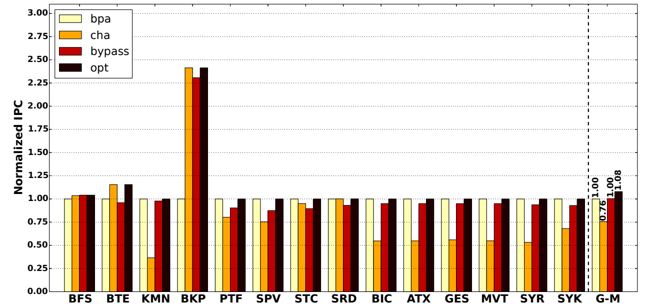

Figure 15: 16KB L1 cache bypassing on Kepler GPU.

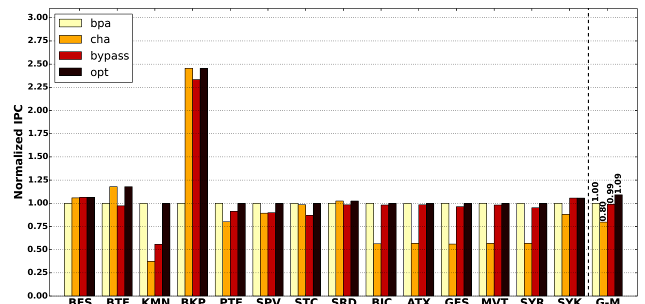

Figure 16: 32KB L1 cache bypassing on Kepler GPU.

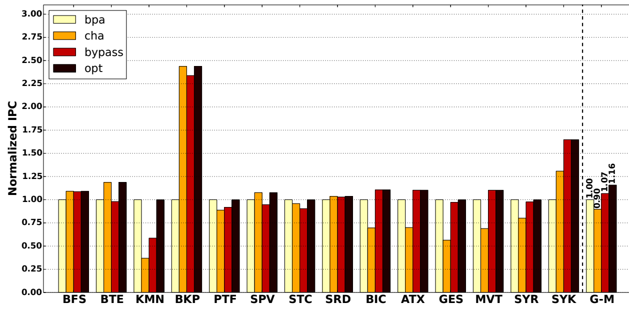

Figure 17: 48KB L1 cache bypassing on Kepler GPU.

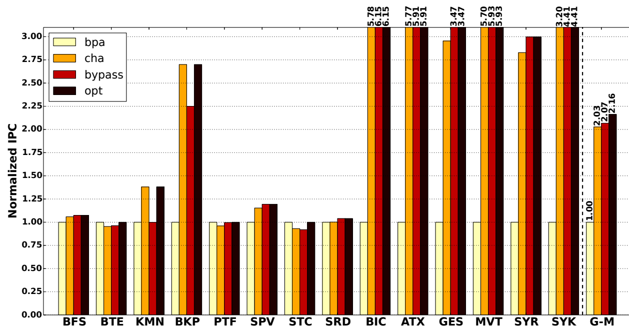

Figure 18: Read-only cache bypassing on Kepler GPU.

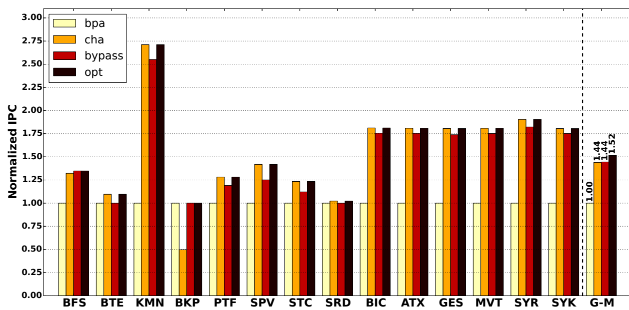

Figure 19: L2 cache bypassing on Kepler GPU.

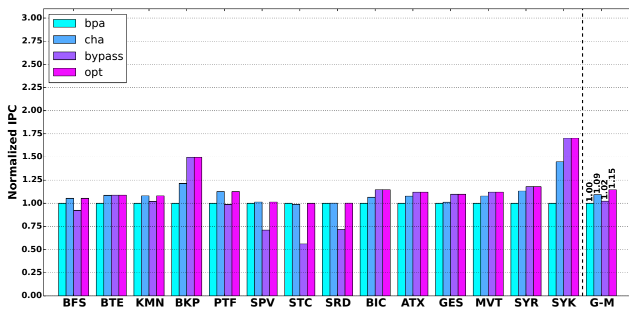

Figure 20: Read-only cache bypassing on Maxwell GPU.

cache and L2 cache bypassing. The results are shown in Figures 20 and 21.

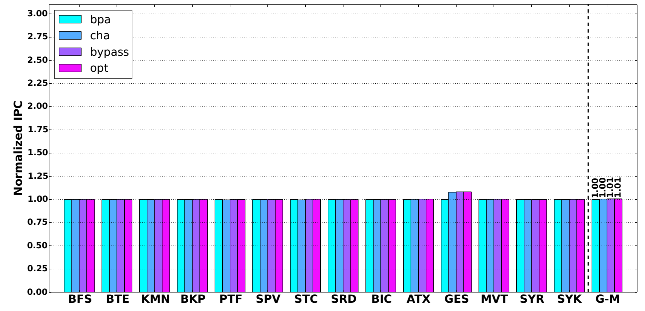

Figure 21: L2 cache bypassing on Maxwell GPU.

Different from Kepler, the read-only cache for Maxwell is not that beneficial, which exhibits a 9% speedup. Moreover, cache bypassing brings only 15% better performance than `bpa` for read-only cache bypassing and almost none for L2 cache. In addition, it should noted that the overhead for cache bypassing is more significant on Maxwell: 13% for read-only cache. We explain the reasons for L2 bypassing results in Section 5.1 and the overhead problem in Section 6.1.

5.1 Performance Analysis Across Platforms

Figure 22 summarizes the geo-mean performance gains for all the applications with all possible caches & cache configurations for the seven GPU platforms in Table 2. As can be seen, for Fermi CC-2.0 and 2.1, cache bypassing is quite effective, especially on large L1 caches and L2 caches. Note that `cha` with 16KB L1 degrades performance by 11% and 15% respectively compare to `bpa`. This explains why from Kepler, L1 cache no longer remains the default datapath for global memory access.

For Kepler CC-3.0, the bars are identical (Kepler-3.0 L1-16K/32K/48K in Figure 22). This is because in Kepler CC-3.0, the L1 cache is only for local memory access [17]. Therefore, bypassing L1 or not does not impact global memory access. For CC-3.5 and 3.7, bypassing works perfectly for read-only caches and L2 caches. Again, L1 cache is detrimental while the bypassing framework eliminates such negative effects effectively.

Regarding Maxwell CC-5.0 and 5.2, bypassing improves performance for read-only cache. However, there is no performance gain on L2. This is because in Maxwell, the "`.cs`" suffix has been abandoned. Therefore, bypass or not generate exactly the same code. We validate this by checking the SASS code — `.cs` and `.ca` produce identical binary file.

5.2 Performance Analysis Across Applications

For applications, regarding their behaviors against threshold variation, we can characterize them into five categories: *bypass-favorite, cache-favorite, cache-congested, cache-insensitive* and *irregular*. For *bypass-favorite* applications, the performance continuously degrades with a higher bypass threshold. This may be due to the rapidly increased L2 traffic induced by the larger L1 cache-line size [33]. `bpa` is the best choice for these applications. Conversely, for *cache-favorite* applications, the performance keeps increasing with a higher threshold. These applications have good locality while the footprints are small enough to be effectively captured by the cache. This condition occurs mostly on L2 and `cha` is the optimal choice. *Cache-congested* applications are those with

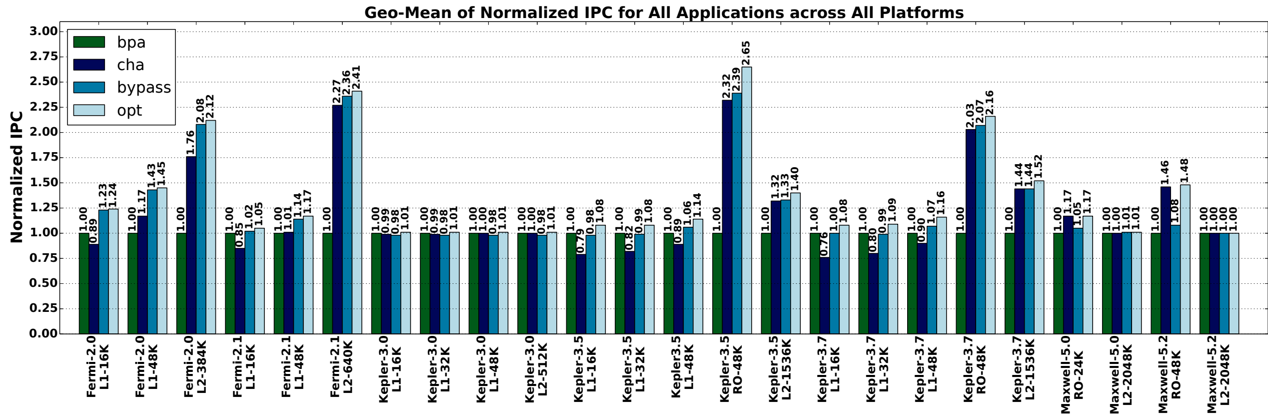

Figure 22: Performance for all applications across all platforms. For the x-ticks, the left column is the major architecture and compute capability of the platform while the right column is the cache type and size.

good locality but experience congestion due to insufficient cache size, such as `bfs` in the case study. The shapes of these applications are convex while the optimal threshold attains in the middle. These applications are the best candidates for cache bypassing. *Cache-insensitive* applications (e.g. `stencil`) have little locality while the overhead from the bypassing framework is quite obvious in the figures. Finally, *irregular* applications show an irregular shape that has no clear trend (e.g. `syrk`). This may be due to the irregularity of the algorithms or datasets. To view the typical figures for each category discussed, please refer to the supplementary file. Note, for the first four **regular** categories, the trend is not very sensitive with the variation of the dataset. Therefore, if we can determine the trend by profiling on a typical dataset, the same option (i.e. `bpa`, `cha` or a certain threshold value) may be applied to other datasets.

5.3 Optimization Suggestions

In addition to the bypassing analysis, we propose several optimization suggestions for general cache utilization:

- In Fermi, if there is no big pressure on shared memory usage, always adopt the 48KB L1 configuration. Otherwise, bypass L1 via ptxas option "*dlcm=cg*" if no bypassing is applied.
- In Kepler, try to use the read-only cache instead of the L1 unless you know it will be beneficial.
- In Kepler and Maxwell, apply the read-only cache bypassing just on the data that are "read-only" in the kernels. Otherwise, you may suffer from performance degradation (e.g. about 6% for Maxwell in our experiments).
- In all architectures, using "__restrict__ const" on read only data reduces register usage (up to half in our observation) and improves code generation quality [21] (e.g. about 16% performance gain for Maxwell L2).

6. DISCUSSION

In this section, we discuss the possibility to reduce bypassing overhead (i.e. predicate register checking per load) via software and hardware approaches. We also clarify *why the proposed cache bypassing design incurs more overhead on Kepler and especially Maxwell than on Fermi.*

6.1 Software Approach

The major reasons for the larger overhead in Kepler and Maxwell than in Fermi, is that after we insert the bypass branches into the *PTX* program, when converting *PTX* into binary, the *ptxas* assembler performs aggressive optimizations, which attempts to combine the many "small divergence" together. In our observation of the *SASS* code, instead of being divergent only at the load operations, the optimized code diverges in much larger code sections and uses completely different registers. This leads to higher register usage and poor instruction cache performance. However, such case is not observed in the code generation for Fermi. Therefore, a direct reaction for reducing overhead is to modify the *SASS* code directly rather than *PTX*. However, there is no official *SASS* assembler available till now and *ptxas* is not open-source. A homemade assembler such as "*maxas*" may help, but is out of the scope of the paper.

Another simple software method is to replicate the whole kernel so that a warp branches from the beginning: *if bypass, a warp executes the copy of kernel with bypassing; otherwise, executes the copy without bypassing.* However, we did not apply this optimization in this paper because: first, it doubles the static code size of the kernel. Second, it may lead to thrashing in the SMs' instruction caches. Please refer to the discussion about "code overlaying" in [34]. Finally, one has to carefully handle the possible interplay between warp branching and TB-wise synchronizations. Nonetheless, we would evaluate this optimization as a future work.

6.2 Hardware Approach

The hardware method is to realize the judging process of bypassing in the cache controller. We use a 5-bit register (32 warps at most). to conserve the bypassing threshold. The register is configured when the kernel is launched. Then, for each memory request, upon it arrives at the cache, its warp index is compared with the threshold register, if less, it is appended to the cache waiting queue, otherwise, it is forwarded to the request queue of the lower memory devices. For example, if bypassing L1, the request is forwarded to the *MRQ* [24] and is later injected into the interconnection network.

Migrating the bypassing functionality into the hardware eliminates the 1-bit predicate register cost per thread as well as the corresponding assessment of it upon each time's

memory access, which improves performance and reduces power. In fact, we implemented this hardware design in GPGPU-Sim [4] using GTX480 (Fermi) architecture with 16KB L1. The simulation results show that the hardware implementation is slightly better than the software regarding both performance and power (2% performance improvement and 2% energy reduction). However, as GPGPU-Sim does not perfectly mimic the behaviors of the real hardware (e.g. based on our previous work [8], Fermi hardware uses an XOR-based hashing in the L1 cache, but such module is not implemented in GPGPU-Sim), there is a big mismatch for some applications (e.g. SSC and BKP) between the simulation outcome and the real hardware measurement (i.e. Figure 12). Therefore, we did not include the figures here but put them in the supplementary file.

7. RELATED WORK

Recently warp-throttling and cache bypassing for enhancing the performance of GPU caches became hot topics [14, 15, 10, 27, 9, 22, 35, 36].

Rogers et al. [14] proposed a cache-conscious wavefront scheduler (CCWS) to limit the number of active wavefronts to be allocated when lost locality was detected. CCWS was later refined as divergence-aware warp scheduling (DAWS) [15], which used a divergence-based cache footprint predictor to assess the L1 cache capacity that was able to capture intra-warp locality within loops. Xie et al. [10] developed a compiler framework to parse the application code and select a set of load operations that bypassing them at L1 could reduce the most L2 cache traffic, based on an ILP or a heuristic optimizer. These operations were then appended with the "*cg*" suffix for bypassing the L1 cache at runtime. The design was tested on a Kepler GTX-680 platform. To compare, their design was a vertical bypass design. The selecting process for bypassing set, as proved in their paper, was an NP-hard problem. Besides, their design was only for L1 cache of Fermi and a small number of Kepler GPUs. Further, L2 traffic reduction did not necessarily lead to the shortest execution time. Very recently, Li et al. [27] proposed another vertical design for GPU L1 cache bypassing. By integrating a locality filter in the L1 cache, memory requests with low reuse or long reuse distance can be excluded from polluting L1. Jia el al. [9] proposed a dynamic hardware approach that bypasses memory load requests when experiencing resource unavailability stalls, particularly cache associativity stalls. While their design might greatly reduce stall waiting, blindly bypassing memory requests whenever there were resource bound might be a bit aggressive, which could hamper performance. The design was runtime resource based which had little relevance to the features of the applications. Chen et al. [22] developed a hardware bypassing mechanism to protect hot cache lines from early eviction based on *lost locality score* detection. Meanwhile, as cache bypassing may lead to congestion at NoC or DRAM, a warp-throttling function for the warp scheduler was supplemented to limit the number of active warps if necessary. Such a design was also runtime hardware based. Mekkat et al. [35] concentrated on CPU-GPU heterogeneous platforms and observed that GPU applications with sufficient thread-level parallelism could tolerate long memory access latency. Therefore, memory requests from GPU threads could bypass LLC while leaving the space for cache sensitive CPU applications. Li et al. [36] implemented a priority-token based hardware design for L1 cache bypassing. In the design, each active warp is allocated with "an additional scheduler status bit". Several "oldest" running warps are granted with high priority while their status bits are set, meaning that only these warps can access the L1 cache. The value of the bit is then appended to each memory request so that the L1 cache is notified.

Most of these schemes, however, concentrated on the architectural design of the memory hierarchy and suggested complicated hardware refinement, which required significant efforts and were not able to **bring instant performance gain to the existing GPUs**. Besides, the validation of the schemes were performed on simulators. As a comparison, our design is purely software and is straightforward to implement. It leverages the reconfigurability of the existing hardware, thus is beneficial to most existing GPUs. Our design can be embedded into the compiler toolchain or encapsulated as a runtime library. Xie et. al. [10] adopted similar cache suffix-based approach as ours. However, as discussed, their bypassing scheme was vertical-based. The search space is much larger. Besides, they focused on L1 only and validated using a single platform GTX-680 (In fact, we are confused about why a Kepler with CC-3.0 can exploit L1.). The very recent work by Li et. al. [36] is a horizontal design. However, it is hardware based that significant area and runtime overhead are introduced: e.g. the additional status bit registers, the extended memory request length, the delay of token management, etc. In addition, reassigning tokens upon each barrier impairs intra-warp locality and may lead to unnecessary inter-warp thrashing. Furthermore, they also concentrated on L1 only and validated using the GPGPU-Sim simulator. However, as discussed in Section 6.2 and the supplementary file, the simulator does not accurately simulate the complete behavior of the GPU caches. Our work confirms that cache bypassing can derive performance on real hardware, in a much simpler software approach that is transparent and adaptive.

8. CONCLUSION

In this paper, we proposed an adaptive cache bypassing framework for GPUs. It used a straightforward approach to throttle the number of warps that could access the three types of GPU caches – L1, L2 and read-only caches, thereby avoiding the fierce cache thrashing of GPUs. Our design is purely software-based thus is able to benefit existing platforms directly. It is easy to implement and is transparent to both the users and the hardware. We validated the framework on seven GPU platforms that covered all GPU generations. Results showed that adaptive bypassing could bring significant speedup over the general cache-all and bypass-all schemes. We also analyzed the performance variation across the platforms and the applications. In addition, we proposed software and hardware approaches to further reduce bypassing overhead and provided several optimization guidelines for the utilization of GPU caches.

8.1 Acknowledgments

We would like to thank the anonymous reviews for their extremely useful comments. Without these comments, the paper could not be improved so significantly. We would also thank Mr. Weifeng Liu from University of Copenhagen and Mrs. Ivan Nosha from Novatte in Singapore for providing some of the GPU platforms and assistance on tests.

References

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. "A Survey of general-purpose computation on graphics hardware". In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library. 2007.

[2] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

[3] W. H. Wen-Mei. *GPU Computing Gems Emerald Edition*. Elsevier, 2011.

[4] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator". In: *ISPASS*. IEEE. 2009.

[5] P. N. Glaskowsky. *Nvidia's Fermi: the first complete GPU computing architecture*. 2009.

[6] J. Nickolls and W. J. Dally. "The GPU computing era". In: *IEEE Micro* 30.2 (2010).

[7] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. "Nvidia Tesla: A unified graphics and computing architecture". In: *Ieee Micro* 28.2 (2008).

[8] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. "A detailed GPU cache model based on reuse distance theory". In: *HPCA*. IEEE. 2014.

[9] W. Jia, K. A. Shaw, and M. Martonosi. "MRPB: Memory request prioritization for massively parallel processors". In: *HPCA*. IEEE. 2014.

[10] X. Xie, Y. Liang, G. Sun, and D. Chen. "An efficient compiler framework for cache bypassing on GPUs". In: *ICCAD*. IEEE. 2013.

[11] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das. "Neither more nor less: Optimizing thread-level parallelism for GPGPUs". In: *PACT*. IEEE Press. 2013.

[12] V. Volkov and J. W. Demmel. "Benchmarking GPUs to tune dense linear algebra". In: *SC*. IEEE. 2008.

[13] Y. Zhang and J. D. Owens. "A quantitative performance analysis model for GPU architectures". In: *HPCA*. IEEE. 2011.

[14] T. G. Rogers, M. O'Connor, and T. M. Aamodt. "Cache-conscious wavefront scheduling". In: *MICRO*. IEEE Computer Society. 2012.

[15] T. G. Rogers, M. O'Connor, and T. M. Aamodt. "Divergence-aware warp scheduling". In: *MICRO*. ACM. 2013.

[16] Z. Zheng, Z. Wang, and M. Lipasti. "Adaptive Cache and Concurrency Allocation on GPGPUs". In: (2013).

[17] Nvidia. *CUDA Programming Guide*. 2015.

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

[19] A. Jog, O. Kayiran, 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: *ACM SIGARCH Computer Architecture News* 41.1 (2013).

[20] Nvidia. *CUDA Best Practice Guide*. 2015.

[21] Nvidia. *Kepler Tuning Guide*. 2015.

[22] 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*. IEEE. 2014.

[23] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser. "Many-core vs. many-thread machines: Stay away from the valley". In: *Computer Architecture Letters* 8.1 (2009).

[24] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc. "Many-thread aware prefetching mechanisms for GPGPU applications". In: *MICRO*. IEEE. 2010.

[25] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. "Orchestrated scheduling and prefetching for GPGPUs". In: *ACM SIGARCH Computer Architecture News* 41.3 (2013).

[26] Nvidia. *PTX: Parallel Thread Execution ISA*. 2015.

[27] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou. "Locality-Driven Dynamic GPU Cache Bypassing". In: *ICS*. ACM, 2015.

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

[29] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter. "Enabling and Exploiting Flexible Task Assignment on GPU Through SM-Centric Program Transformations". In: *ICS*. ICS '15. ACM, 2015.

[30] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. Hwu. "Parboil: A revised benchmark suite for scientific and commercial throughput computing". In: *Center for Reliable and High-Performance Computing* (2012).

[31] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. "Mars: a MapReduce framework on graphics processors". In: *PACT*. ACM. 2008.

[32] 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 (InPar)*. IEEE. 2012.

[33] W. Jia, K. A. Shaw, and M. Martonosi. "Characterizing and improving the use of demand-fetched caches in GPUs". In: *ICS*. ACM. 2012.

[34] M. Bauer, S. Treichler, and A. Aiken. "Singe: leveraging warp specialization for high performance on GPUs". In: *ACM SIGPLAN Notices* 49.8 (2014).

[35] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai. "Managing shared last-level cache in a heterogeneous multicore processor". In: *PACT*. IEEE Press. 2013.

[36] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder. "Priority-based cache allocation in throughput processors". In: *HPCA*. IEEE. 2015.