An Efficient Compiler Framework for Cache Bypassing on GPUs

Yun Liang, Xiaolong Xie, Guangyu Sun, and Deming Chen

***Abstract*—Graphics processing units (GPUs) have become ubiquitous for general purpose applications due to their tremendous computing power. Initially, GPUs only employ scratchpad memory as on-chip memory. Though scratchpad memory benefits many applications, it is not ideal for those general purpose applications with irregular memory accesses. Hence, GPU vendors have introduced caches in conjunction with scratchpad memory in the recent generations of GPUs. The caches on GPUs are highly configurable. The programmer or compiler can explicitly control cache access or bypass for global load instructions. This highly configurable feature of GPU caches opens up the opportunities for optimizing the cache performance. In this paper, we propose an efficient compiler framework for cache bypassing on GPUs. Our objective is to efficiently utilize the configurable cache and improve the overall performance for general purpose GPU applications. In order to achieve this goal, we first characterize GPU cache utilization and develop performance metrics to estimate the cache reuses and memory traffic. Next, we present efficient algorithms that judiciously select global load instructions for cache access or bypass. Finally, we present techniques to explore the unified cache and shared memory design space. We integrate our techniques into an automatic compiler framework that leverages parallel thread execution instruction set architecture to enable cache bypassing for GPUs. Experiments evaluation on NVIDIA GTX680 using a variety of applications demonstrates that compared to cache-all and bypass-all solutions, our techniques improve the performance from 4.6% to 13.1% for 16 KB L1 cache.**

***Index Terms*—Cache bypassing, compiler, graphics processing unit (GPU), performance.**

I. Introduction

With the continuing evolution of heterogenous computing platforms that consist of graphics processing units (GPUs) and CPUs, GPUs are increasingly used for high-performance embedded computing. Due to their massively parallel architecture, GPUs benefit a variety of embedded applications including imaging, audio, aerospace, military, and medical applications [1]–[3]. Indeed, recent years have also seen a rapid adoption of GPUs in mobile devices like smartphones. Mobile devices typically use system-on-a-chip (SoC) that integrates GPUs with CPUs, memory controllers, and other application-specific accelerators. The major SoCs with integrated GPUs available in the market include NVIDIA Tegra series with low power GPU, Qualcomm's Snapdragon series with Adreno GPU, and Samsung's Exynos series with ARM Mali GPU.

Despite the high-computing power of GPUs, performance optimization of GPUs is challenging. The achieved performance speedup critically depends on memory subsystem [4], [5]. In early GPUs, software-managed scratchpad memory (also known as shared memory) was employed as the on-chip memory to hide the memory access latency. Data allocation to scratchpad memory can be explicitly controlled by the programmer or automatically by the compiler. Scratchpad memory benefits certain applications with predictable data access patterns, but it is not appropriate for applications with irregular access patterns. For these applications, they naturally prefer cache instead of scratchpad memory. Ideally, the optimal memory hierarchy should combine the benefits of both scratchpad memory and cache. Indeed, in the recent generations of GPUs, GPU vendors have introduced cache in conjunction with scratchpad memory to improve the memory performance. For example, both NVIDIA Fermi and Kepler have a unified cache (L1 cache) and scratchpad memory. These architectures also have a L2 cache to further exploit data localities.

For both CPUs and GPUs, caches can effectively hide the data access latency by exploiting the program localities. However, GPU caches are quite distinct from CPU caches in terms of design and utilization. Meanwhile, the caches on GPUs are highly configurable. GPU architecture provides interfaces for the programmer or compiler to explicitly control the L1 cache access or bypass for global load instructions. Cache bypassing is very beneficial for applications with memory accesses that are scattered or have no data localities as it can help to improve memory efficiency [6].

Although cache bypassing can potentially improve GPU performance, it is a challenge for the programmer. Given a program that consists of n global load instructions, the number of possible cache bypassing solutions is exponential (2^n). These global load instructions cannot be considered

Manuscript received November 28, 2014; revised January 24, 2015; accepted February 16, 2015. Date of publication April 21, 2015; date of current version September 17, 2015. This work was supported in part by the National Natural Science Foundation of China under Grant 61300005, and in part by CFAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. This paper was recommended by Associate Editor J. Henkel. *(Corresponding author: Yun Liang.)*

Y. Liang and G. Sun are with the Center for Energy-Efficient Computing and Applications, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China, and also with Collaborative Innovation Center of High Performance Computing, National University of Defense Technology, Changsha 410073, China.

X. Xie is with the Center for Energy-Efficient Computing and Applications, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China.

D. Chen is with the University of Illinois at Urbana-Champaign, Champaign, IL 61801, USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2015.2424962

0278-0070 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

in isolation ($O(n)$) as they are usually dependent on each other (i.e., data reuse or conflict). Obviously, it is infeasible for the programmer to manually use the cache bypassing interface and explore the huge design space exhaustively. More importantly, recent work has shown that GPU caches have counter-intuitive performance tradeoffs [7]. In particular, neither cache-all or bypass-all global load instructions is optimal; if the cache bypassing is not done right, it may seriously hurt the performance. Thus, it is very important to develop automatic compiler techniques for cache bypassing on GPUs.

The unified memory architecture on GPUs (L1 cache and shared memory) is configurable. For example, NVIDIA Fermi and Kepler architectures allow the programmers a choice over its L1 cache and shared memory partition. More specifically, on NVIDIA GTX680, the 64 KB unified memory can be configured as either a 16 KB cache and 48 KB shared memory, a 32 KB cache and 32 KB shared memory, or a 48 KB cache and 16 KB shared memory. The trend in the unified memory design will allow more fine-grained partition choices between cache and shared memory [8]. The shared memory capacity determines the number of parallel executing threads while the L1 cache capacity determines the cache performance of a single thread. Thus, the programmers have to strike the right balance between the L1 cache and shared memory.

In this paper, we propose an efficient compiler framework for cache bypassing on GPUs that aims to improve the performance for general purpose GPU applications. We first characterize GPU cache utilization and develop performance metrics to accurately estimate the cache reuses and memory traffic. In particular, we use light-weight profiling to characterize each global load instruction, data reuse, load efficiency, and memory traffic. We also use static analysis to identify frequently used memory access patterns for accurate load efficiency computation. Next, we develop algorithms that judiciously select global load instructions for cache access or bypass. One algorithm is based on integer liner programming (ILP) and the other one is a heuristic. Finally, we determine a good partition between the L1 cache and shared memory. Our framework leverages the parallel thread execution (PTX) instruction set architecture (ISA). Experimental results on NVIDIA GTX680 show that our compiler framework for cache bypassing can effectively improve the overall performance for GPU applications.

This paper contributes to the state-of-the-art in GPU optimization with the following.

1) *Compiler Framework:* We develop an efficient compiler framework for cache bypassing on GPUs. It automatically analyzes GPU code and implements the optimized cache bypassing solution by leveraging the PTX ISA.
2) *Cache Bypassing Algorithms:* We develop two algorithms for cache bypassing optimization. One algorithm is based on ILP and the other is an efficient heuristic. Both algorithms are based on traffic reduction graph, which captures the data reuse, conflicts between global load instructions and load efficiencies.
3) *Unified Cache and Shared Memory Exploration:* We demonstrate the tradeoff in the cache and shared memory capacity and develop regression model based performance metrics to guide the design space exploration and determine the best partition between L1 cache and shared memory.
4) *Experimental Evaluation:* We use a variety of applications for evaluation. Experiments on the NVIDIA GTX680 show that compared to cache-all solution, our cache bypassing technique improves the average cache benefits from 4.6% to 13.1% for 16 KB L1 cache. The performance speedup of our cache bypassing with unified cache and shared memory exploration technique is up to 2.62×.

A preliminary version of this paper was reported in [9]. In this paper, we propose load efficiency computation techniques for frequently used access patterns and explore unified cache and shared memory space.

This paper is organized as follows. In Section II, we provide some background on GPUs and present a motivational example for our cache bypassing study. In Section III, we introduce our compiler framework and the involved analysis components. In Sections IV and V, we detail the characterization and optimization components. Section VI presents the experimental results. Section VII discusses the related work. Finally, Section VIII concludes this paper.

II. Background and Motivation

This section provides the background details and motivation of this paper. We use the NVIDIA Kepler GTX680 GPU architecture and CUDA terminology in this paper. But our techniques are equally applicable to other GPUs with caches and the OpenCL programming model.

A. Background

State-of-the-art GPUs are many-core architectures. Based on NVIDIA terminology, a GPU is composed of multiple streaming multiprocessors (SMs), which in turn is composed of multiple streaming processors (SPs). For example, the NVIDIA GTX680 used in this paper has eight SMs, each of which has 192 SPs. Thus, there are 1536 cores in total. Each SM has private registers, which are shared among the threads running on it. Threads are organized into thread blocks. A thread block is scheduled to execute on one of the SMs. CUDA provides synchronization barrier routine for the threads within a thread block. Upon a synchronization point, threads have to wait until all the other threads reach the barrier. On one SM, threads are scheduled in units of warps (32 threads). The threads in a warp execute in an SIMD style.

Fig. 1 shows the memory hierarchy of recent generations of GPUs with caches. Each SM is equipped with caches in conjunction with scratchpad memory. The low latency on-chip scratchpad memory is also called shared memory for GPUs. For example, each SM of the NVIDIA Fermi and Kepler architectures contains a configurable 64 KB on-chip memory, which is shared by scratchpad memory and L1 data cache [10], [11]. The programmer can choose how much storage to devote to the L1 cache versus shared memory (16 versus 48 KB, 32 versus 32 KB, and 48 versus 16 KB). Shared memory is allocated per thread block. The same shared memory can be accessed by all the threads within the same thread block. The data stored in

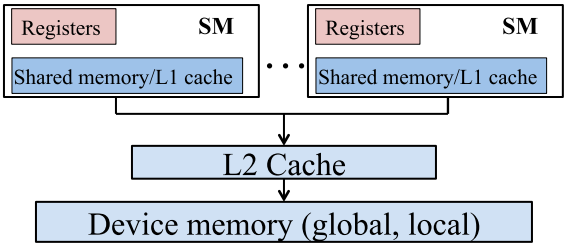


Fig. 1. GPU memory hierarchy.

TABLE I
PARAMETERS OF GTX680

Parameters	Values
Compute capability	3.0
Number of SMs	8
Number of SPs per SM	192
L2 cache size	512 KB (32-byte block)
L1 cache size	16, 32, 48 KB (128-byte block)
Shared memory size	48, 32, 16 KB
Maximum threads per SM	2048
Maximum thread blocks per SM	16
Number of 32-bit registers per SM	64K
Graphics core clock	1006 Mhz

on-chip shared memory can be accessed much faster than that stored in off-chip global memory. All the SMs share a unified L2 cache.

Global and local memories reside in cached device memory. In other words, the accesses to data in global and local memory have to go through the two-level cache hierarchy. Local memory is used as a per-thread private memory space for register spills, function calls, and automatic array variables. For GPU applications, the majority of cached data accesses are from/to global memory. Thus, in this paper, we focus on the cache bypassing for the global memory. Table I describes the architecture details of NVIDIA GTX680 (Kepler architecture) used in this paper.

NVIDIA Fermi and Kepler architectures provide interfaces to explicitly control the L1 cache access or bypass for global load instructions. In particular, the programmer or the compiler can configure the L1 cache in either coarse- or fine-grained manner. In a coarse-grained manner, all the global load instructions are cached or bypassed. We refer to this as "cache-all" or "bypass-all," respectively. This is controlled by using compilation flags (-dlcm=ca or -dlcm=cg). In a fine-grained manner, each individual global load instruction can choose either cache access or cache bypass (detailed program interface in Section III). L1 caches on different SMs are not coherent while L2 cache is coherent across all the SMs on the chip. Finally, current NVIDIA GPUs do not cache global store data in L1 cache because L1 caches are not coherent for global data. Thus, global stores ignore L1 caches, and discard any L1 cache line if it is matched. This behavior is not configurable in the current GPU architecture. Thus, in this paper we only focus on global memory loads.

B. Motivation

Here, we present the performance speedup potential through cache bypassing on GPUs and the need of automatic compiler support. We use the kernel particle filter from Rodinia benchmark suite [12] as our case study. There are totally 14 load instructions in particle filter. To limit the design space, we only choose six global load instructions with high-access frequencies to be cache bypassing candidates. Fig. 2 shows how the performance speedup varies with the cache bypassing solutions for this sub-space. The results are normalized to cache-all (e.g., cache all the six chosen global load instructions). The horizontal axis represents the subset of design space ($2^6 = 64$ solutions), the left-most point represents cache-all and the right-most point represents bypass-all. The performance can be increased to 1.18× by using the best bypassing solution as shown in Fig. 2. The other eight loads have small impact on performance as they not frequently used. If we consider the entire design space (all the 14 loads), the best performance speedup can be increased only to 1.21×. The experiments are performed on NVIDIA Kepler GTX680 and we use NVIDIA profiler to collect the kernel performance.

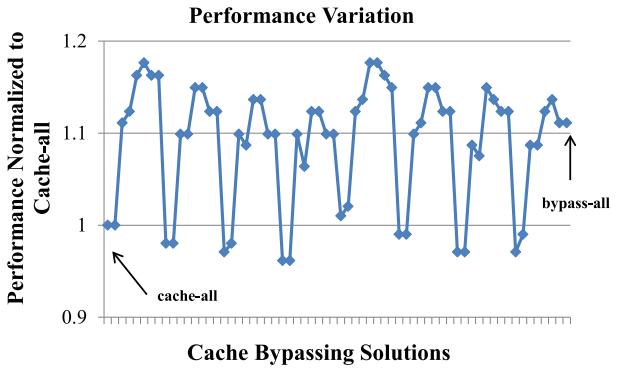


Fig. 2. Performance variation of particle filter with different cache bypassing solutions.

As shown in Fig. 2, the performance speedup critically depends on the cache bypassing solutions. Neither cache-all or bypass-all ensures good performance. More importantly, there is a considerable performance speedup potential by exploiting the cache bypassing interface. Meanwhile, the cache bypassing optimization necessitates an automatic compiler framework as it is infeasible to manually explore such a huge design space for GPU applications.

III. COMPILER FRAMEWORK

Fig. 3 presents our compiler framework for cache bypassing on GPUs. The CUDA code is first precompiled to PTX code, which is CUDAs intermediate representation used in NVIDIA CUDA compiler. Our compiler framework takes the unmodified PTX code as input and outputs the optimized CUDA binaries. The framework involves three components: characterization, optimization, and instrumentation. Initially, characterization component collects the data access, reuse, and load efficiency through light-weight profiling. Then, optimization component determines cache access or bypass for every global load instruction. Finally, instrumentation component implements the optimized solution determined by the optimization component by leveraging the PTX ISA. The characterization component is also assisted with the instrumentation component for profiling.

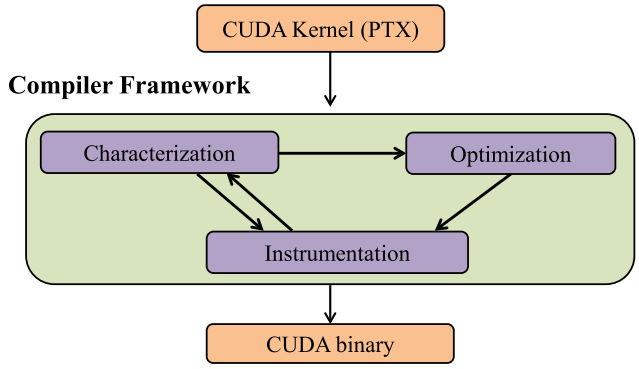


Fig. 3. Compiler framework for cache bypassing on GPUs.

```
1 ld.global.f32 %f2, [%rd23+0];
2 st.shared.f32 [%rd14+0], %f2;
3 .loc 14 82 0
4 ld.global.f32 %f3, [%rd19+0];
5 st.shared.f32 [%rd15+0], %f3;
```

Listing. 1. Original PTX code

```
1 ld.global.cg.f32 %f2, [%rd23+0];
2 st.shared.f32 [%rd14+0], %f2;
3 .loc 14 82 0
4 ld.global.ca.f32 %f3, [%rd19+0];
5 st.shared.f32 [%rd15+0], %f3;
```

Listing. 2. Modified PTX code

The characterization and optimization components are detailed in Sections IV and V, respectively. Here, we describe the details of the instrumentation component. Our implementation leverages the PTX ISA. Global memory loads in PTX are in the following format:

$$\text{ld.global.}L1\_cache\_option\ dst, src$$

L1_cache_option has six possible values

$$ca, cg, cs, lu, cv, empty$$

among which we focus on *ca*, *cg*, *empty*, which represent cache access, cache bypass, and default setting.

For each global load, we can explicitly control its cache access or bypass by using different *L1_cache_option*. More clearly, we implement cache access using option *ca* and implement cache bypass using option *cg*. After all these changes, we need to update the PTX section size and embed it into CUDA binary. We illustrate the PTX code instrumentation using an example (Listings 1 and 2). After the PTX code instrumentation, the first global load (line 1 in Listing 2) bypasses the L1 cache while the last global load (line 4 in Listing 2) accesses the L1 cache.

IV. Characterization Component

GPU architecture is quite distinct from CPU architecture. In this section, we first characterize the distinct features in terms of cache utilization on GPUs and then present the performance metrics and load efficiency computation for cache bypassing optimization component.

A. Cache Block Size

For GPUs, the cache block sizes are different for L1 and L2 caches [10], [11]. In particular, L1 cache has 128 bytes block size while L2 cache has 32 bytes block size. Depending on the requested data size and data access patterns, data transfers are separated into one or more cache blocks. More clearly, when L1 cache is used, the hardware issues transfers of 128 bytes; otherwise, the hardware issues transfers of 32 bytes. However, not all the transferred data in a cache block are useful, that is to say, the global load efficiency is less than or equal to 100%. According to the NVIDIA profiler, the global load efficiency is defined as

$$\text{efficiency} = \frac{\text{useful data}}{\text{transferred data}}.$$

We use E_{on} to denote the global load efficiency when L1 cache is accessed and E_{off} to denote the global load efficiency when L1 cache is bypassed. For example, for an aligned 16-byte data request, E_{on} is $16/128 = 12.5\%$ and E_{off} is $16/32 = 50\%$. Hence, L1 cache bypassing is very beneficial for applications with scattered memory accesses, because a memory access that fetches 128 bytes for L1 cache significantly wastes the memory bandwidth. In this case, L1 cache bypassing can help to improve the load efficiency and thus reduce memory traffic.

B. Data Locality

GPUs are many-core architectures. Thousands of threads may execute and share the L1 cache simultaneously on the same SM. As a result, cache contention among threads is more significant on GPUs than on CPUs. In the extreme case, if all the threads in a thread block execute together in lock-step style, then it is less likely for caches to exploit temporal locality. However, in the real GPU hardware, threads execute in warps, and the scheduler may execute a few instructions for the current warp before it switches to the next warp. Thus, GPU caches still exploit both spatial and temporal localities. We define two types of data localities on GPUs.

1) *Intrawarp Spatial/Temporal Locality:* The threads within the same warp access the same cache line.
2) *Interwarp Spatial/Temporal Locality:* The threads within different warps access the same cache line.

The threads in a warp are executed in an SIMD style. Intrawarp spatial locality refers to the memory coalescing for GPUs. The continuous memory accesses from the threads within a warp lead to coalesced accesses. Intrawarp spatial locality is determined by the data access pattern of the warp and the coalescing policy of the GPU architecture. Intrawarp temporal locality exists because the threads within a warp may execute a few instructions before switching to the next warp and the neighboring instructions tend to have data localities. In fact, intrawarp temporal locality is very significant for the GPU benchmarks we studied in the experiments. For example, different fields of a data structure are accessed consecutively in

the code. Finally, interwarp locality is possible as different warps may access the same data (e.g., array [tid % 32]).

The above data localities are closely related. More importantly, both intrawarp and interwarp localities depend on the warp scheduling policy of the real hardware. Thus, it is difficult to analyze them separately and statically. In this paper, we use light-weight profiling to characterize the data localities as shown in the next section.

C. Performance Metrics

Let the GPU kernel have N global load instructions. We order the global load instructions according to their program order. We use ld_i to denote the ith global load instruction. We rely on the instrumentation component described in Section III to modify the GPU code and use the NVIDIA profiler to collect the following metrics.

1) *access$_i$:* The number of L1 cache accesses for ld_i. This number is obtained using profiler by bypassing L1 cache for all the global load instructions except ld_i.
2) *hit$_i$:* The number of L1 cache hits for ld_i. This number is obtained as a by-product of access$_i$.
3) *hit$_{i,j}$:* The number of L1 cache hits for ld_i and ld_j together. This number is obtained using profiler by bypassing L1 cache for all the global load instructions except ld_i and ld_j.

Let us define $\text{gain}_{i,j}$

$$\text{gain}_{i,j} = \text{hit}_{i,j} - \left(\text{hit}_i + \text{hit}_j\right).$$

We use $\text{gain}_{i,j}$ to measure the data reuses or conflicts between ld_i and ld_j. Note that, $\text{gain}_{i,j}$ may be either positive or negative. If $\text{gain}_{i,j}$ is positive, it means ld_i and ld_j have data reuses, and we should cache them together to exploit the data localities between them; otherwise, ld_i and ld_j conflict with each other, and we should bypass either one of them, or both of them. We could use $\text{gain}_{i,j}$ to estimate L1 cache hit rate. However, L1 cache hit rate does not predict performance well as demonstrated in prior work [7]. High-L1 hit rate does not guarantee high performance as fetching L1 cache block (128 bytes) leads to high-L2 cache traffic. Hence, we extend our metrics with awareness of cache block size and use L2 cache traffic as performance indicator. For ld_i, we use $T_{\text{on}}(\text{ld}_i)$ ($T_{\text{off}}(\text{ld}_i)$) to denote the L2 cache traffic when L1 cache is accessed (bypassed) for ld_i

$$T_{\text{on}}(\text{ld}_i) = (\text{access}_i - \text{hit}_i) \times L1\_block\_size.$$

Depending on the data access patterns, one data transfer from L1 cache may be separated into n ($1 \leq n \leq 4$) transfers from L2 cache when L1 cache is bypassed. Note that, n may not always be $4 = 128/32$. For example, to transfer an aligned 64-byte data, we need one 128-byte transfer if L1 cache is cached; otherwise, we need two 32-byte transfers if L1 cache is bypassed (L2 cache block is 32 bytes). For this case, $n = 2$. However, NVIDIA profiler does not give the exact number of transferred L2 blocks for each global load. Instead, we compute $T_{\text{off}}(\text{ld}_i)$ as follows:

$$T_{\text{off}}(\text{ld}_i) = \frac{\text{access}_i \times L1\_block\_size \times E_{\text{on}}(\text{ld}_i)}{E_{\text{off}}(\text{ld}_i)}$$

where $E_{\text{on}}(\text{ld}_i)$ ($E_{\text{off}}(\text{ld}_i)$) is the load efficiency of ld_i when L1 cache is accessed (bypassed). We present the computation of $E_{\text{on}}(\text{ld}_i)$ ($E_{\text{off}}(\text{ld}_i)$) in Section IV-D.

Definition 1 (Traffic Reduction Graph): Let traffic reduction graph $\text{TG} = (V, E)$ be a weighted and complete graph, where node $v_i \in V$ represents ld_i. Nodes and edges are weighted using function W. The weight of node v_i, $W(v_i) = T_{\text{off}}(\text{ld}_i) - T_{\text{on}}(\text{ld}_i)$; the weight of edge $e(v_i, v_j)$, $W(e(v_i, v_j)) = \text{gain}_{i,j} \times L1\_block\_size$.

The weight function W estimates the L2 cache traffic reduction of cache access over cache bypass. $W(v_i)$ or $W(e(v_i, v_j))$ could be either positive or negative. If it is positive, it means L1 cache access can reduce L2 cache traffic by exploiting data localities; otherwise, it means L1 cache access can increase the L2 cache traffic due to cache conflicts or low-load efficiency. The negative nodes and edges prefer cache bypassing.

In this paper, we use profiling to characterize the data locality, load efficiency, and L2 cache traffic. The profiling runs very fast (see Section VI). More importantly, GPU kernels usually are frequently called for many times. Thus, the profiling overhead is very low compared to the kernel runtime. For the applications with dynamic behaviors, a more detailed profiling may be necessary. However, for embedded system applications, their program behaviors are more predictable and thus tend to be stable across inputs. In Section VI, we will demonstrate the robustness of this profiling based method across different inputs. More clearly, we use different inputs for profiling and evaluation and show high-performance speedup.

D. Computation of Load Efficiency

Global loads are mainly used to load data from arrays stored in global memory. For each global load, its load efficiency is defined as the ratio of the useful data to the transferred data. Based on this definition, we know that the load efficiency is determined by the access pattern of the load, cache line size, and memory coalescing policy. Thus, different loads may have different load efficiencies as they may use different access patterns. In order to compute the load efficiency, we can use the NVIDIA profiler. However, the existing profiler has very rudimentary support for load efficiency profiling. Specifically, it only gives the overall load efficiency for the entire application. In this section, we present an automatic analysis that identifies a few frequently used access patterns and compute their load efficiencies statically.

Our analysis parses the PTX code of the kernel and identifies three commonly used array access patterns shown in Table II. The first pattern refers to streaming data access with a stride. The second pattern refers to partial sharing access pattern, where a few threads share the same data. The third pattern refers to full sharing, where all the threads share the same data. For each access pattern, Table II gives the computation of load efficiency.

We use the examples in Listing 3 to illustrate the load efficiency computation for each access pattern. *bfs_kernel* is from benchmark BFS. In this example, the global load (*graph_nodes*[$i + tid$]) is executed in a loop. When executing

TABLE II
ANALYZABLE ACCESS PATTERNS

Array Access Patterns	Load Efficiency
$V + tid \times C_0$ ($C_0 \geq 1$)	$\frac{min(max(Bsize/(Esize \times C_0), 1), 32) \times Dsize}{Bsize}$
$V + tid/C_0$ ($C_0 \geq 1$)	$\frac{min(max(Esize \times 32/C_0, 1), Bsize) \times Dsize}{Esize \times Bsize}$
V	$min(\frac{Dsize}{Bsize}, 1)$

```
1  __global__ void bfs_kernel(Node*
       graph_nodes, ....)
2  {
3    int tid = blockIdx.x*THREAD_NUM +
         threadIdx.x;
4    for(int i = 0;i < End;i+=Stride)
5    {
6      int tmp = graph_nodes[i+tid].
           edges
7      ...
8    }
9  }
10 __global__ void srk_kernel(float *
       d_data, int num,...)
11 {
12  int tid=blockIdx.x*blockDim.x+
        threadIdx.x;
13  int index=tid/16;
14  float bound=d_data[num+index];
15  ...
16 }
17 __global__ void spm_kernel(float *
       dst_vector, int idx)
18 {
19     ...
20     int j = dst_vector[idx];
21     ..
22 }
```

Listing. 3. Examples for Load Efficiency Computation

the load instructions, the data transfer unit size is the cache block size (Bsize). For each cache block transfer, the useful data size is the product of the number of threads and the requested data size per thread (Dsize). Threads on GPUs are executed in the unit of warps (32 threads). Thus, the number of threads involved in the transfer of one cache block is

$$\min(\max(\text{Bsize}/(\text{Esize} \times C_0), 1), 32)$$

where Esize represents the array element size. In *bfs_kernel*, its array element size is the size of data structure *Node* and its access stride $C_0 = 1$. The requested data size per thread (Dsize) is the size of an integer (variable tmp on line 6). Then, the load efficiency can be computed using the formula as shown in Table II. Note that, for each load ld_i, we compute its $E_{\text{on}}(\text{ld}_i)$ and $E_{\text{off}}(\text{ld}_i)$ using Bsize = 128 and 32, respectively.

Similarly, we can compute the load efficiencies for the other two patterns. For example, *srk_kernel* from benchmark SRK in Listing 3 is an instance of the second pattern, where every 16 threads share the same data (e.g., $tid/16$); *spm_kernel* from benchmark SPM in Listing 3 is an instance of the third pattern, where all the threads load from the same memory location. For the access patterns not in Table II, we refer them as unknown access patterns. For the unknown access patterns, we use the global load efficiency of the program given by the profiler as their load efficiency.

V. OPTIMIZATION COMPONENT

A. Problem Formulation

Given N global load instructions, we could select a subset of global load instructions for cache bypassing. Thus, there exist 2^N cache bypassing solutions. For each candidate solution, we could use compiler framework that automatically generates the compilable PTX code, runs the code and empirically evaluates the performance and chooses the best one. However, obviously this approach is infeasible for complex and large programs.

Our solution is developed based on the traffic reduction graph. We consider the traffic reduction graph as if it were an exact representation of L2 cache traffic reduction. Given a complete subgraph $G' = (V', E')$ of TG $= (V, E)$ where $V' \subseteq V$ and $E' \subseteq E$, we define the traffic reduction by caching all the global load instructions in G' as

$$T(G') = \sum_{v \in V'} W(v) + \sum_{e \in E'} W(e).$$

Therefore, we formulate a problem that maximizes the L2 cache traffic reduction as follows.

Problem 1 (Traffic Reduction Maximization): Given traffic reduction graph $G = (V, E)$, find a complete subgraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$, such that $T(G')$ is maximized.

Theorem 1: Traffic reduction maximization problem is NP hard.

Proof: We show a reduction from maximal clique problem [13]. Given an instance of maximal clique problem, i.e., a graph $G = (V, E)$, we construct an instance of traffic reduction maximization problem. Let TG $= (V', E', W')$, where $V' = V$ and TG is a complete graph. Thus, $E \subseteq E'$. Let $\forall v \in V', W(v) = 1$, $\forall e \in E, W(e) = 1$, and

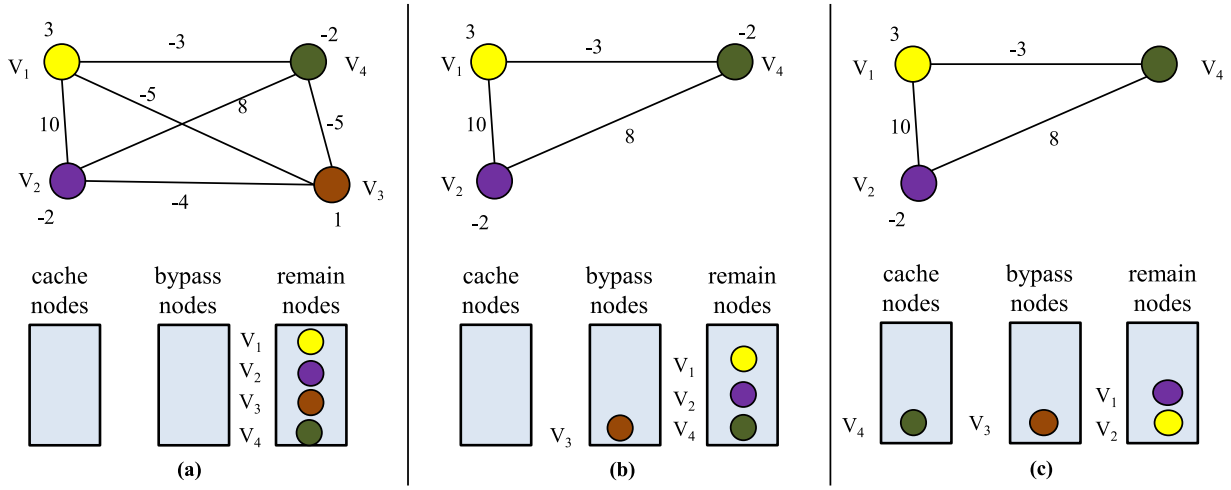


Fig. 4. Illustration of our heuristic algorithm. (a) Original traffic reduction graph. (b) and (c) Updated traffic reduction graph.

$\forall e \in E' \setminus E, W(e) = -\infty$. This reduction is polynomial time. Then, to solve the maximal clique problem for $G = (V, E)$, we just need to solve the traffic reduction maximization problem for TG $= (V', E', W')$. Thus, traffic reduction maximization problem is NP hard. ■

B. ILP Formulation

We develop an ILP formulation to solve the traffic reduction maximization problem exactly. In practice, the ILP solution can be applied to programs with small number of global load instructions.

For a traffic reduction graph TG $= (V, E)$, our optimization objective is to maximize

$$\sum_{v_i \in V} W(v_i) \times N_{v_i} + \sum_{e(v_i, v_j) \in E} W\big(e\big(v_i, v_j\big)\big) \times M_{v_i, v_j}$$

where N_{v_i} and M_{v_i, v_j} are 0-1 decision variable

$$N_{v_i} = \begin{cases} 1 & \text{cache ld}_i \\ 0 & \text{bypass ld}_i. \end{cases}$$

We have the following constraints:

$$M_{v_i, v_j} = N_{v_i} \times N_{v_j}.$$

We linearize the above equations as follows:

$$\begin{aligned} N_{v_i}, M_{v_i, v_j} &= 0 \text{ or } 1 \\ M_{v_i, v_j} &\le N_{v_i} \\ M_{v_i, v_j} &\le N_{v_j} \\ M_{v_i, v_j} &\ge N_{v_i} + N_{v_j} - 1. \end{aligned}$$

For each global load ld_i, it is cached if $N_{v_i} = 1$; otherwise, it is bypassed.

C. Heuristic Algorithm

ILP formulation is not scalable to large programs. Thus, we also develop an efficient polynomial-time heuristic. Algorithm 1 presents the details of our heuristic. It is an iterative algorithm. In each iteration, for every global load, we first evaluate its potential traffic reduction if it is cached together with other cached and remaining global loads (line 6). We select the one with the minimal traffic reduction (line 7). Then, we add its own traffic reduction. If the overall traffic reduction is positive, it is cached; otherwise, it is bypassed. If a node is bypassed, then it is deleted from the traffic reduction graph; otherwise it is kept in the traffic reduction graph for evaluation of the remaining nodes.

Fig. 4 shows an example of the heuristic algorithm. The traffic reduction graph consists of four nodes (four global loads).

Algorithm 1: Heuristic Approach

Input: $TG = (V, E)$
Output: V_{cache}, the set of cached global loads,
V_{bypass}, the set of bypassed global loads

1 $V_{remain} = V$; //Initialization
2 **while** $|V_{remain}| > 0$ **do**
3 //find the $min\_v \in V_{remain}$ with minimal traffic reduction with the others;
4 $min\_T = INFINITE$; $min\_v = NULL$;
5 **foreach** $v_i \in V_{remain}$ **do**
6 $$T_{other}(v_i) = \sum_{v_j \in V_{cache}} W\big(e\big(v_i, v_j\big)\big) + \sum_{v_k \in V_{remain} \wedge v_k \neq v_i} W(e(v_i, v_k))$$
if $T_{other}(v_i) \le min\_T$ **then**
7 $min\_T = T_{other}(v_i)$; $min\_v = v_i$;
8 $T = T_{other}(min\_v) + W(min\_v)$;
9 **if** $(T \le 0)$ **then**
10 //bypass it;
11 delete (min_v) from TG;
12 add min_v to V_{bypass};
13 **else**
14 //cache it;
15 add min_v to V_{cache};
16 delete (min_v) from V_{remain};

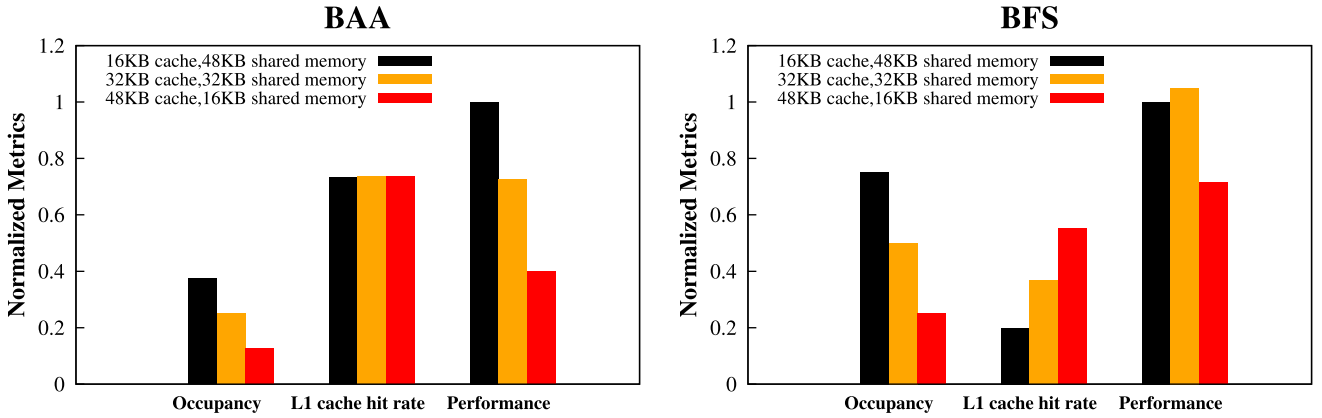


Fig. 5. Occupancy, L1 cache hit rate, and performance for different L1 cache and shared memory partitions when all the global loads use cache. The performance is normalized to 16 KB L1 cache and 48 KB shared memory.

The nodes and edges are weighted based on the traffic reduction metrics. In the first iteration of the algorithm, the node V_3 is selected as it has the minimal traffic reduction with others ($-5-5-4=-14$); and V_3 is bypassed as its overall traffic reduction is negative ($-14+1=-13$). In the second iteration, we choose V_4 and it is cached. Note that, the traffic reduction graph is updated only when the selected node is bypassed.

D. Unified Memory Exploration

In Sections V-B and V-C, we have presented algorithms to select beneficial load instructions for cache bypassing for a specific cache size. The state-of-the-art GPU architectures (Fermi and Kepler) feature with configurable unified memory, which is shared by the L1 cache and shared memory. For example, the NVIDIA GTX680 allows three partition choices between the L1 cache and shared memory (16 versus 48 KB, 32 versus 32 KB, and 48 versus 16 KB). However, the compiler chooses 16 KB L1 cache and 48 KB shared memory for all the applications by default. It leaves the partition challenge to the programmers.

Large shared memory capacity improves the thread level parallelism (TLP) if the shared memory usage is the resource bottleneck. High TLP provides more opportunities to enhance performance by hiding the long memory latency through thread context switch. Hence, one would suggest to use maximum shared memory size (48 KB). However, maximum shared memory size leads to minimal L1 cache size (16 KB). Small L1 cache size may decrease L1 cache hit rate and thus increase the L2 cache traffic. If the increase in TLP does not compensate the increase in cache traffic, then the overall performance might be degraded. Furthermore, high TLP may aggravate cache contention among threads and lead to higher L1 and L2 cache misses. Hence, large shared memory with high TLP does not always guarantee good performance for GPU applications. The best L1 cache and shared memory partition may vary for different applications depending on their shared memory usage and data localities.

To evaluate different L1 cache and shared memory partitions, we need to model both TLP and cache performance. On one hand, for different shared memory capacity, we use occupancy to model the TLP. Occupancy is a metric to measure the application and platform parallelism. It is defined as the ratio of the number of simultaneously active threads to the maximum number of threads supported on one SM [14]. Occupancy is determined by the resource usage per thread block including registers, shared memory, and the architecture limits of specific GPU architecture. The NVIDIA NVCC compiler provides an interface to obtain the occupancy at compile time [14]. Let us assume the maximum number of threads supported on one SM is 1024 and a thread block with 256 threads requires 16 KB of shared memory. If the shared memory capacity is 16 KB, then only one thread block can execute simultaneously. This leads occupancy as 0.25 (256/1024). If we increase the shared memory capacity to 48 KB, then three thread blocks can execute simultaneously and this leads to occupancy as 0.75.

On the other hand, we use L1 cache hit rate to model the cache performance, it could be obtained by trace-driven simulation or analysis models [15], [16].

Fig. 5 presents the occupancy, L1 cache hit rate, and performance for different shared memory and cache partitions for the BAA and BFS applications. In this example, we do not use cache bypassing for any global loads. In other words, all the loads access the cache. The best shared memory and cache partition are different for these two applications as they are different in terms of shared memory usage, data localities, and thread numbers. For BAA, 16 KB cache and 48 KB shared memory is the best as it leads to high occupancy. In addition, for BAA, 16 KB cache gives good cache performance as the program working set can fit into it. For BFS, 32 KB cache and 32 KB shared memory is the best as it achieves a balance between the occupancy and cache performance.

We use Occ to denote the achieved occupancy. $0 < \text{Occ} \leq 1$. We use HitRate to denote the L1 cache hit rate when all the loads are cached. $0 < \text{HitRate} \leq 1$. To model the performance effect of thread level parallelism and cache, we build a regression based performance model using Occ and HitRate as follows:

$$\text{Perf} = \beta_0 + \beta_1 \times \text{Occ} + \beta_2 \times \text{HitRate} + \beta_3 \times \text{Occ} \times \text{HitRate}. \quad (1)$$

TABLE III
BENCHMARK CHARACTERISTICS

Benchmark	Source	Description	Shared Memory (KB)	Input 1	Input 2
backprop (BAA)	Rodinia [13]	backward propagation algorithm	16	32768	65536
bfs (BFS)	Rodinia [13]	breadth first search	16	64K nodes	1M nodes
euler3d (EUF)	Rodinia [13]	redundant flux computation	16	193K	0.2M
kmeans (KMI)	Rodinia [13]	k-means clustering	16	97M	25M
particle filter (PFFL)	Rodinia [13]	particle filter algorithm	4	128*128*1000	128*128*10000
srad prepare (SRP)	Rodinia [13]	data preparation for anisotropic diffusion	2	50*0.5*502*458	100*0.5*502*458
srad reduce (SRR)	Rodinia [13]	1D Vector reduce	4	50*0.5*502*458	100*0.5*502*458
srad kernel (SRS)	Rodinia [13]	speckle reducing anisotropic diffusion	0	50*0.5*502*458	100*0.5*502*458
spmv (SPM)	Parboil [19]	sparse matrix-vector multiplication	0	2M	60M
mri-gridding (MGR)	Parboil [19]	magnetic resonance imaging gridding	0	1M	2.6M
3dc kernel(3DC)	Polybench [20]	3D convolution	0	512*512*512	256*256*256
syrk kernel(SRK)	Polybench [20]	symmetric rank-k operations	0	2048*512	1024*512
gemm kernel(GEM)	Polybench [20]	matrix multiplication	0	50k	100k
aligntypes(ALT)	SDK [21]	data alignment evaluation	0	30k	50k
mri-q (MRQ)	SDK [21]	magnetic resonance image reconstruction	0	32*32*32	64*64*64

We include Occ $\times$ HitRate factor as the TLP and cache performance are correlated. Note that, Perf is not designed for the absolute performance, but for relative comparison of different shared memory and L1 cache partitions.

We fit the unknown parameters (β_0, β_1, β_2, and β_3) using linear least square regression method. More clearly, we choose six cache or shared memory usage sensitive benchmarks (BAA, BFS, EUF, KMI, SRP, and SRR) from Table III as the training set. For each benchmark, we collect its performance metrics for all the three possible L1 cache and shared memory partitions. Then, we fit the parameters (β_0, β_1, β_2, and β_3) using the training data. The achieved mean square error is 0.035. Note that parameters (β_0, β_1, β_2, and β_3) are platform dependent. This training and fitting only need to be done once on one platform.

Now, with (1), we can determine a good shared memory and L1 cache partition, then we use the algorithms in Section V-C to choose the global loads for cache bypassing for this partition.

VI. EXPERIMENTAL EVALUATION

A. Experiments Setup

We evaluate our technique on NVIDIA GTX680 (Kepler architecture). The hardware details of GTX680 are presented in Table I. We test a variety of benchmarks from benchmark suite Rodinia [12], Parboil [17], Polybench [18], and NVIDIA GPU Computing SDK [19]. The tested benchmarks are general-purpose GPU applications with diverse characteristics including thread structure, computation, and memory access patterns. The benchmark details are shown in Table III. For each benchmark, our compiler framework performs a lightweight profiling to characterize the data locality and load efficiency, builds the traffic reduction graph, invokes our cache bypassing optimization algorithms, and modifies the CUDA PTX code to reflect the optimized cache bypassing solution. In our experiment, for each benchmark, we provide two inputs to evaluate the robustness of our profiling-based techniques against different inputs as shown in Table III.

We implement both optimal and heuristic algorithms. For the optimal solution, we use MOSEK [20] to solve the ILP problem. NVIDIA GTX680 has configurable unified memory design. Through CUDA library call, it can be configured to 16 KB cache and 48 KB shared memory, 32 KB cache and 32 KB shared memory, or 48 KB cache and 16 KB shared memory. In the following, we first show the performance of our cache bypassing on different cache sizes in Section VI-B. Then, we present the input sensitivity study and unified memory exploration results in Sections VI-C and VI-D, respectively. Finally, we show the efficiency (run-time) of our compiler framework. In our experiments, all the performance data are measured through NVIDIA profiler.[1,2]

B. Performance Results

We compare four solutions: bypass-all, cache-all, heuristic, and ILP. In cache-all solution, all the global load instructions go through the L1 cache; in bypass-all solution, all the global load instructions bypass the L1 cache. We try all the three possible size of caches (16, 32, and 48 KB) on NVIDIA GTX680. Fig. 6 presents the performance results for different size of caches. The performance is normalized to the bypass-all solution. In this experiment, we use input 1 in Table III for both profiling and evaluation.

First of all, neither cache-all or bypass-all solution guarantees good performance for all the benchmarks. Such coarse-grained solutions may be good for small benchmarks that have loads with similar access patterns (e.g., KMI), but most likely give bad performance for benchmarks that have loads with diverse access patterns (e.g., 3DC and PFFL). In contrast, our heuristic performs consistently well across all the benchmarks. The performance speedup of our cache bypassing technique is up to 2.62$\times$. More clearly, for 16 KB cache, heuristic improves the performance by 13.1% on average while cache-all improves the performance by only 4.6% on average; for 32 KB cache, heuristic improves the performance by 19.1% on average while cache-all improves the performance by 13.0% on average; for 48 KB cache, heuristic improves the performance by 23.4% on average while cache-all improves

[1]PFFL is tested only using 16 KB cache as its memory allocation is unsuccessful for 32 and 48 KB caches.

[2]For applications BAA, BFS, EUF, and KMI, we use the versions with shared memory.

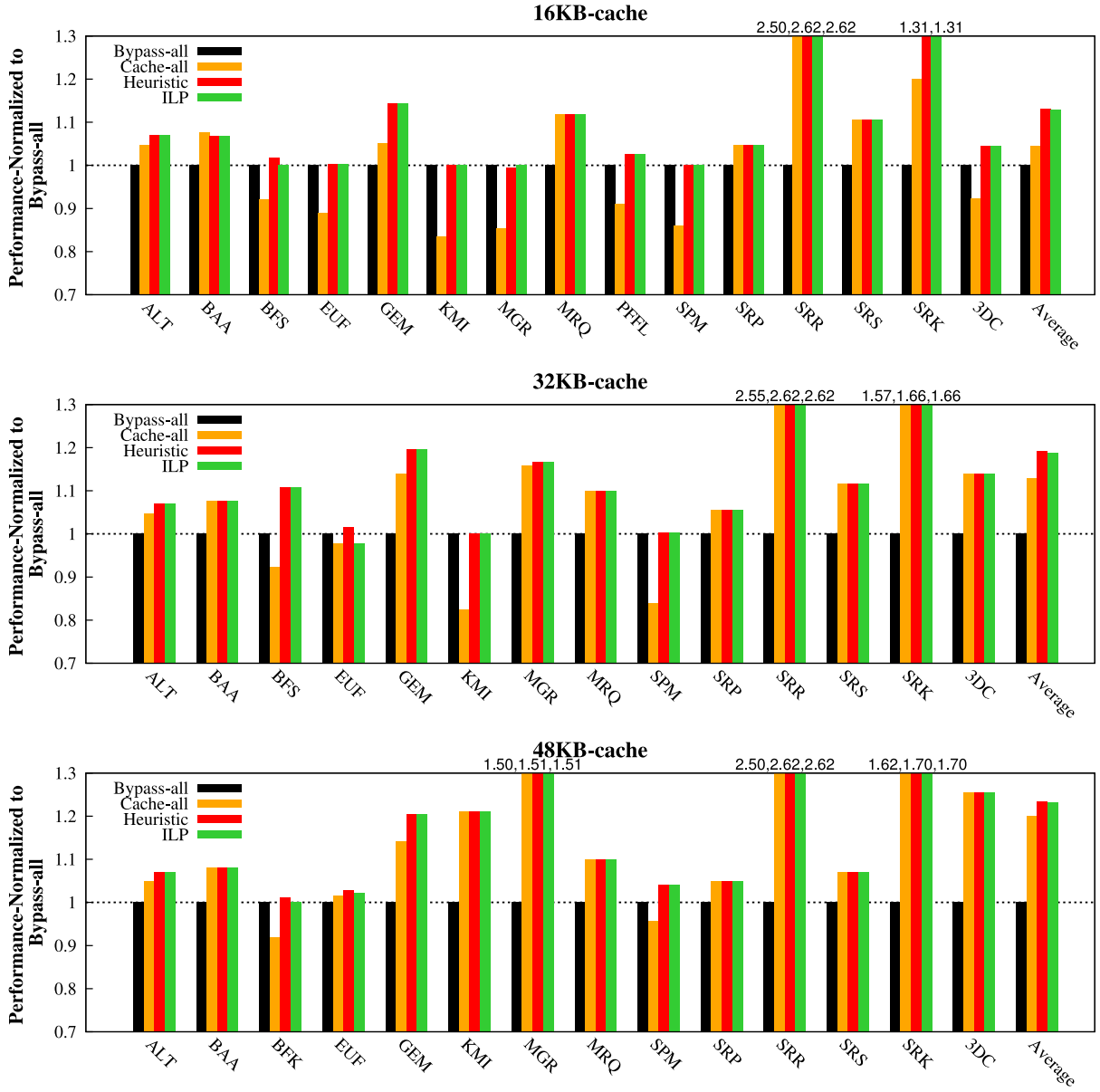


Fig. 6. Performance comparison for different size of caches.

the performance by 20.1% on average. The average value is computed based on geometric mean.

The performance improvement of heuristic solution linearly increases as the cache size increases. This is because a larger cache offers more opportunities to exploit data localities than a smaller cache. We also notice that the speedup of heuristic over cache-all 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 application 3DC, its working set fit into 32 and 48 KB caches. Hence, in these two settings, heuristic solution does not give any speedup for 3DC compared to cache-all solution. However, large caches do not solve all the problems. Designing large caches implies less space for other components. For example, NVIDIA Fermi and Kepler architectures use unified cache and shared memory designs. In this unified design, cache size increase leads to shared memory size decrease. However, shared memory size decrease may hurt the performance as the number of active thread blocks might be reduced. We will demonstrate this in Section VI-D.

Our heuristic and ILP solution return the same results for most of the benchmarks. However, there are a few cases that heuristic is slightly better than ILP solution. This is because for those benchmarks the load efficiencies of some loads are not analyzable statically and approximation using the global efficiency is inaccurate (Section IV-D). Therefore, it is possible that ILP solution results in a sub-optimal solution in practice. But overall heuristic and ILP solutions perform consistently well across all the benchmarks and cache settings.

C. Input Sensitivity

In previous section, we use the same input (input 1) for profiling and evaluation. Here, we evaluate the sensitivity of

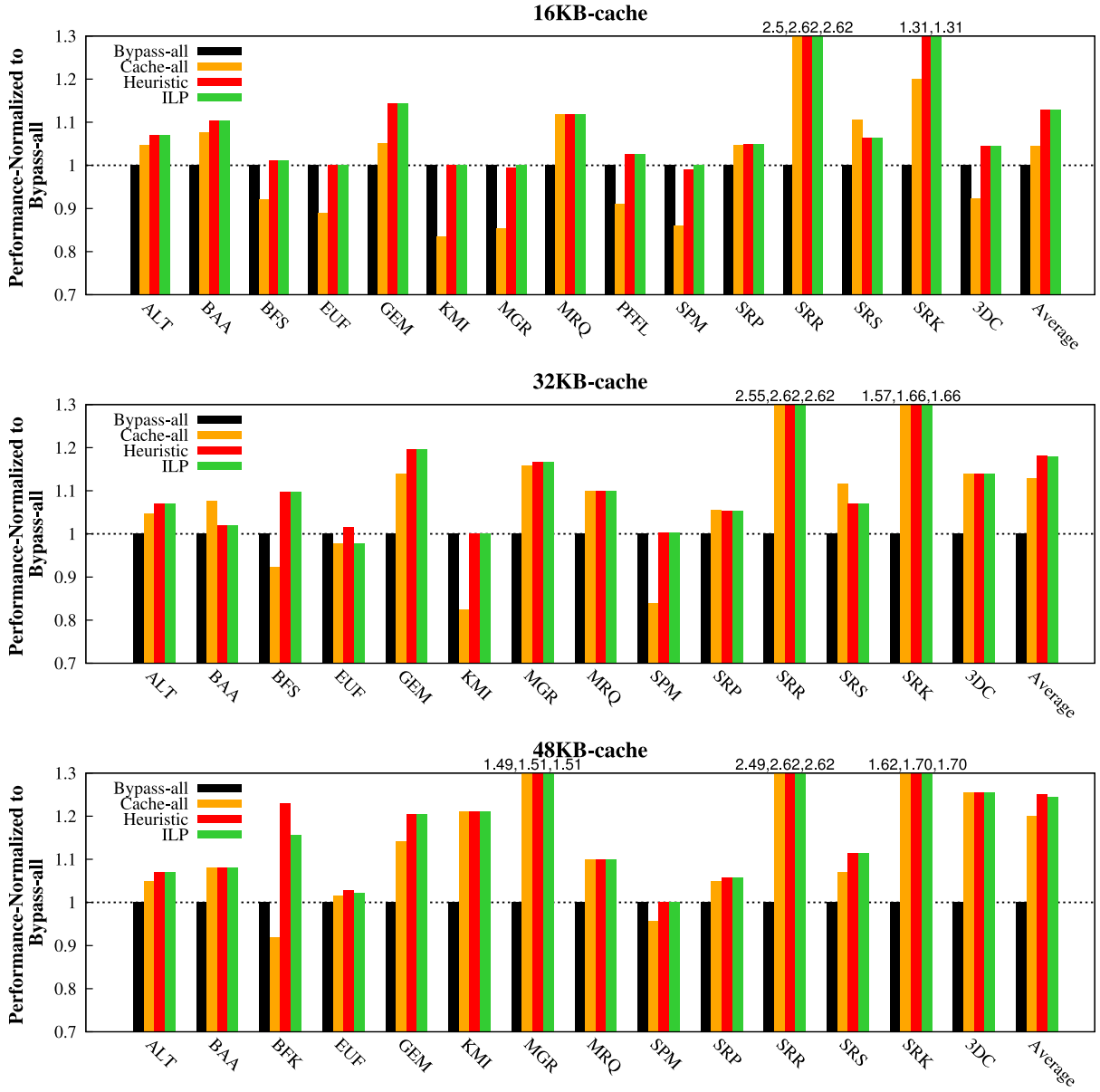


Fig. 7. Input sensitivity study for different size of caches.

our profile-based analysis across different inputs. More clearly, we profile the application using input 1 and evaluate the cache bypassing using input 2 shown in Table III. The performance result is shown in Fig. 7. The performance is normalized to the bypass-all solution.

As shown, for most of benchmarks, high-performance speedup is still maintained. On average, heuristic improves the performance by 12.9%, 18.2%, and 25.2% for 16, 32, and 48 KB cache, respectively. High-performance improvement is still achieved because different inputs tend to stay stable in terms of the behaviors of global loads.

D. Unified Memory Exploration

Though NVIDIA Kepler GTX680 features with configurable unified memory design, by default the NVIDIA NVCC compiler chooses 16 KB cache and 48 KB shared memory for all the benchmarks. However, different applications have different behaviors and thus may prefer different unified memory partitions. Fig. 8 shows the performance of three different unified memory partitions for six benchmarks. For example, benchmarks BAA and EUF prefer 16 KB cache and 48 KB shared memory partition. It is because for these two benchmarks each thread block uses 16 KB shared memory as shown in Table III and thus they can execute three thread blocks concurrently with 48 KB shared memory, but only one thread block with 16 KB shared memory. Conversely, for benchmark SRS, 48 KB cache and 16 KB shared memory partition is the best as SRS does not use any shared memory.

In Section V-D, we propose a regression based performance model to compare different cache and shared memory partitions. Our model considers both TLP and cache performance. Fig. 8 compares the actual performance with the estimated performance returned by the model. As stated in Section V-D, the model is not intended to accurately predict performance, but just to compare different shared memory

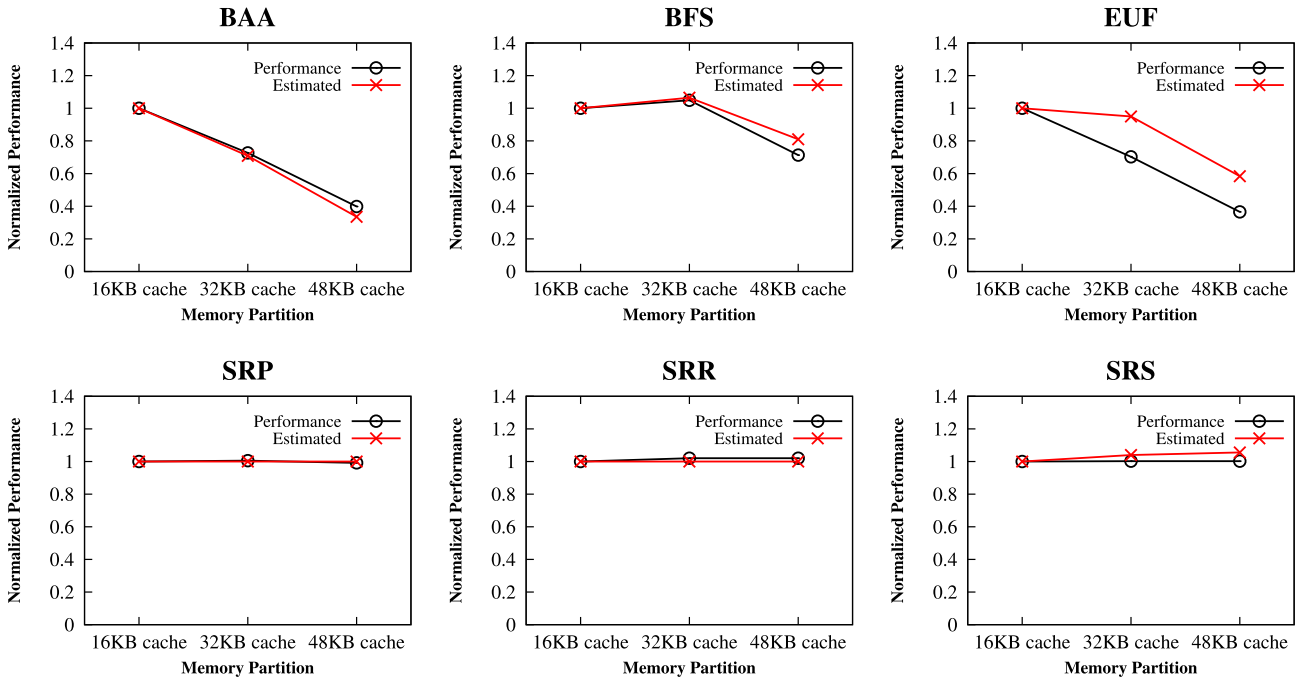


Fig. 8. Comparison of actual performance and estimated performance. Shared memory and cache share the 64 KB space. 16, 32, and 48 KB caches imply 48, 32, and 16 KB shared memory, respectively.

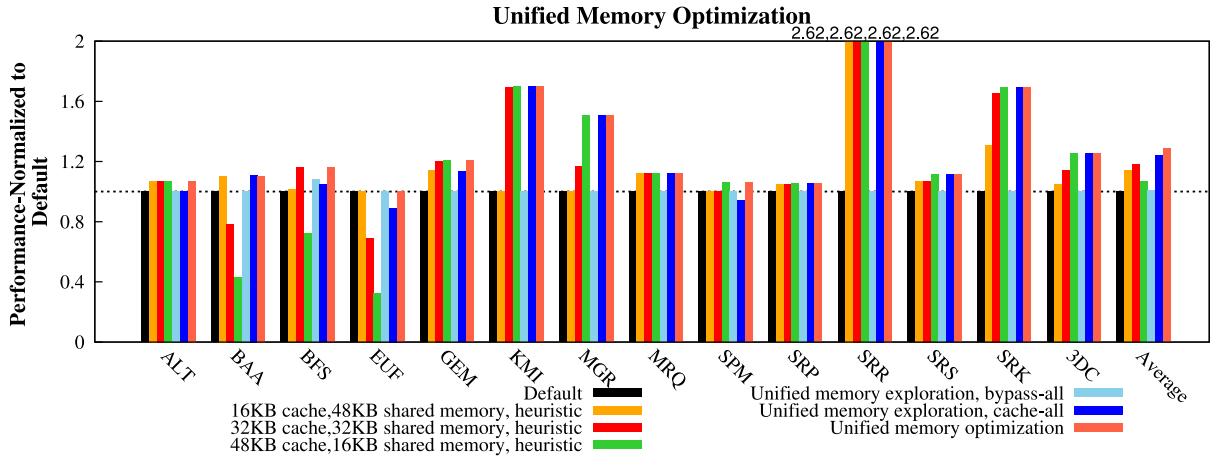


Fig. 9. Performance speedup of the unified memory exploration with different cache bypassing solutions. Performance is normalized to the default setting (16 KB cache and 48 KB shared memory and all the global loads are bypassed).

and L1 cache partitions. As shown in Fig. 8, our model actually captures the performance trend and predicts the best cache and shared memory partition. Next, we combine our unified memory exploration with the heuristic cache bypassing algorithm. In other words, we first use our unified memory exploration to select a good cache and shared memory partition, and then apply our cache bypassing heuristic algorithm to further improve the performance. We refer this as unified memory optimization.

We first compare our unified memory optimization with three possible cache and shared memory partitions with heuristic cache bypassing algorithm separately. For each possible cache and shared memory partition, we test it for all the benchmarks. Fig. 9 shows the comparison. The performance is normalized to the default setting (16 KB cache and 48 KB shared memory partition and cache-all loads). As shown, our unified memory optimization performs consistently well across all the benchmarks. In contrast, the other partitions only benefit a subset of applications. For example, 48 KB cache and 16 KB shared memory partition is a good choice for applications with big memory footprint MGR and SRK, but turns out to be a bad choice for applications with big shared memory usage (e.g., BAA and BFS). Overall, our unified memory optimization achieves 28.3% speedup on average.

We also combine unified memory exploration with cache-all and bypass-all and compare them with our unified memory optimization. The comparison results are shown in Fig. 9.

TABLE IV
COMPILER FRAMEWORK EXECUTION TIME

Benchmark	Runtime (sec)		
	16KB	32KB	48KB
ALT	0.79	0.77	0.77
BAA	1.19	1.103	1.112
BFS	0.44	0.42	0.41
EUF	35.26	35.43	35.34
GEM	0.42	0.44	0.42
KMI	0.04	0.05	0.04
MGR	0.33	0.33	0.33
MRQ	19.67	19.52	19.54
PFFL	1.37	-	-
SPM	0.71	0.71	0.67
SRP	0.06	0.06	0.06
SRR	0.10	0.10	0.10
SRS	0.59	0.59	0.59
SRK	0.99	0.98	1.01
TDC	0.64	0.65	0.65

We observe that coarse-grained bypassing solutions (cache-all and bypass-all) do not perform well even coupled with unified memory exploration. Our unified memory optimization synergistically combines unified memory exploration and fine-grained cache bypassing, and thus achieves better performance improvement.

E. Efficiency

Our compiler framework runs very efficiently, Table IV shows the execution time of our compiler framework with heuristic algorithm for various size of caches. For all the benchmarks, it only takes a few seconds to complete.

VII. RELATED WORK

A. GPU Performance Optimization

Although GPUs promise high performance, tuning GPUs for high performance is not a trivial task [4]. Both analytical performance models [21], [22] and optimization techniques are developed. The state-of-the-art GPU performance optimization techniques focus on automatic data movement, data layout transformation, thread and warp throttling, control flow divergence, register allocation, multitasking, and power and aging optimization [23]–[27]. However, none of above works targets cache bypassing for GPUs.

B. Cache Bypassing for GPUs

There are a few recent studies that explore cache bypassing for GPUs. Runtime cache bypassing with extra hardware supports are proposed in [28]–[30]. As an alternative to hardware approach, static cache bypassing technique has been developed in [7]. More clearly, Jia *et al.* [7] presented a characterization and optimization study for GPU caches. Their characterization study demonstrates that on GPUs L1 cache hit rate does not correlate with performance. However, they analyze the data access patterns manually and thus this is not scalable to large applications. They also assume there is no data reuse between global load instructions and different iterations of the same global load instructions. In contrast, the goal of this paper is to develop an automatic compilation framework that can systematically model both temporal and spatial localities and select the beneficial global loads for cache bypassing. Finally, our technique also explores the unified cache and shared memory partition design space.

C. Cache Bypassing for CPUs

Cache bypassing has been widely used for CPU caches to alleviate cache pressure. Both hardware and compiler bypassing techniques are proposed in [31] and [32]. However, these techniques mainly use cache hit rate as performance metrics to guide the cache bypassing optimization. Hence, these techniques are not applicable to GPUs as cache hit rate does not correlate well with performance on GPUs as demonstrated in [7].

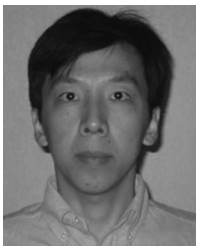
VIII. CONCLUSION

Nowadays, heterogenous computing platforms that consist of CPUs and GPUs are widely adopted for high-performance embedded computing. Recently, caches are also included in modern GPUs. GPU caches allow fine-grained cache bypassing for each global load instruction. In this paper, we develop an efficient compiler framework for cache bypassing on GPUs. Our compiler framework can automatically analyze the GPU code and optimize the code through bypassing the load instructions with low-data reuse, low efficiency, or high-cache conflicts with others. Experiments on NVIDIA GTX680 show that our techniques improve the average cache benefits to 13.1%, 19.1%, and 23.4% for 16, 32, and 48 KB caches, respectively.

REFERENCES

[1] V. Bertacco *et al.*, "On the use of GP-GPUs for accelerating compute-intensive EDA applications," in *Proc. Conf. Design Autom. Test Europe (DATE)*, Grenoble, France, 2013, pp. 1357–1366.
[2] Y. Liang *et al.*, "Real-time implementation and performance optimization of 3D sound localization on GPUs," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Dresden, Germany, Mar. 2012, pp. 832–835.
[3] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.
[4] S. Ryoo *et al.*, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. 13th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, Salt Lake City, UT, USA, Jan. 2008, pp. 73–82.
[5] Y. Kim and A. Shrivastava, "CuMAPz: A tool to analyze memory access patterns in CUDA," in *Proc. 48th Design Autom. Conf. (DAC)*, Jun. 2011, pp. 128–133.
[6] C.-J. Wu *et al.*, "SHiP: Signature-based hit predictor for high performance caching," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit. (Micro)*, Porto Alegre, Brazil, Dec. 2011, pp. 430–441.
[7] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *Proc. 26th ACM Int. Conf. Supercomput. (ICS)*, Venice, Italy, Jun. 2012, pp. 15–24.
[8] 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 *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit. (Micro)*, Vancouver, BC, Canada, Dec. 2012, pp. 96–106.
[9] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, San Jose, CA, USA, 2013, pp. 516–523.
[10] NVIDIA. *Fermi GPUs*. [Online]. Available: http://www.nvidia.com/object/fermi-architecture.html

[11] NVIDIA. *Kepler GPUs*. [Online]. Available: http://www.nvidia.com/object/nvidia-kepler.html
[12] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Character. (IISWC)*, Austin, TX, USA, Oct. 2009, pp. 44–54.
[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. New York, NY, USA: McGraw-Hill, 2001.
[14] NVIDIA. *Occupancy Calculator*. [Online]. Available: http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html
[15] X. E. Chen and T. Aamodt, "Modeling cache contention and throughput of multiprogrammed manycore processors," *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 913–927, Jul. 2012.
[16] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A detailed GPU cache model based on reuse distance theory," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Orlando, FL, USA, Feb. 2014, pp. 37–48.
[17] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Center Rel. High-Perform. Comput., Univ. Illinois Urbana-Champaign, Champaign, IL, USA, Tech. Rep. IMPACT-12-0, Mar. 2012.
[18] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Proc. Innov. Parallel Comput. Conf. (InPar)*, San Jose, CA, USA, May 2012, pp. 1–10.
[19] *NVIDIA GPU Computing SDK*. [Online]. Available: http://developer.nvidia.com/gpu-computing-sdk
[20] *Mosek*. [Online]. Available: http://www.mosek.com/
[21] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. 36th Annu. Int. Symp. Comput. Archit. (ISCA)*, Austin, TX, USA, Jun. 2009, pp. 152–163.
[22] 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 *Proc. 15th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, Bangalore, India, 2010, pp. 105–114.
[23] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Washington, DC, USA, 2012, pp. 72–83.
[24] Y. Liang, Z. Cui, K. Rupnow, and D. Chen, "Register and thread structure optimization for GPUs," in *Proc. 18th Asia South Pac. Design Autom. Conf. (ASP-DAC)*, Yokohama, Japan, Jan. 2013, pp. 461–466.
[25] Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An accurate GPU performance model for effective control flow divergence optimization," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Shanghai, China, May 2012, pp. 83–94.
[26] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang, "Run-time technique for simultaneous aging and power optimization in GPGPUs," in *Proc. 51st Annu. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2014, pp. 1–6.
[27] Y. Liang, H. Huynh, K. Rupnow, R. Goh, and D. Chen, "Efficient GPU spatial-temporal multitasking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 3, pp. 748–760, Mar. 2014.
[28] W. Jia, K. A. Shaw, and M. Martonosi, "MRPB: Memory request prioritization for massively parallel processors," in *Proc. 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Orlando, FL, USA, 2014, pp. 272–283.
[29] X. Chen *et al.*, "Adaptive cache management for energy-efficient GPU computing," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Minneapolis, MN, USA, 2014, pp. 343–355.
[30] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for GPUs," in *Proc. 21th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Burlingame, CA, USA, 2015, pp. 76–88.
[31] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proc. 41st Annu. IEEE/ACM Int. Symp. Microarchit. (Micro)*, Lake Como, Italy, Nov. 2008, pp. 222–233.
[32] Y. Wu *et al.*, "Compiler managed micro-cache bypassing for high performance EPIC processors," in *Proc. 35th Annu. ACM/IEEE Int. Symp. Microarchit. (MICRO)*, Istanbul, Turkey, Nov. 2002, pp. 134–145.



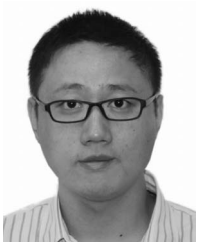
Yun Liang received the B.S. degree in software engineering from Tongji University, Shanghai, China, in 2004, and the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2010.

He was a Research Scientist with the Advanced Digital Science Center, University of Illinois at Urbana-Champaign, Champaign, IL, USA, from 2010 to 2012. He has been an Assistant Professor with the School of Electronics Engineering and Computer Science, Peking University, Beijing, China, since 2012. His current research interests include graphics processing unit architecture and optimization, heterogeneous computing, embedded system, and high-level synthesis.

Dr. Liang was a recipient of the Best Paper Award from the International Symposium on Field-Programmable Custom Computing Machines in 2011 and best paper award nominations from the International Conference on Hardware/Software Codesign and System Synthesis in 2008 and the Design Automation Conference in 2012. He serves as a Technical Committee Member for the Asia South Pacific Design Automation Conference (ASPDAC), the Design Automation and Test in Europe, the International Conference on Compilers Architecture and Synthesis for Embedded System, and the International Conference on Parallel Architectures and Compilation Techniques. He was the TPC Subcommittee Chair for the ASPDAC in 2013.

Xiaolong Xie received the B.S. degree from the Institute of Microelectronics, Peking University, Beijing, China, in 2013, where he is currently pursuing the Ph.D. degree with the Center for Energy-Efficient Computing and Applications.

His current research interests include compiler-level and architecture-level optimization of graphics processing unit memory hierarchies for general-purposed applications.



Guangyu Sun received the B.S. degree in electronic engineering and the M.S. degree in micro-electronics from Tsinghua University, Beijing, China, in 2003 and 2006, respectively, and the Ph.D. degree in computer science from Pennsylvania State University, State College, PA, USA, in 2011.

He is currently an Assistant Professor with the Center for Energy Efficient Computing and Applications, Peking University, Beijing. His current research interests include computer architecture, electronic design automation, and very large-scale integration design.

Deming Chen received the B.S. degree in computer science from the University of Pittsburgh, Pittsburgh, PA, USA, in 1995, and the M.S. and Ph.D. degrees in computer science from the University of California, Los Angeles, Los Angeles, CA, USA, in 2001 and 2005, respectively.

He has been an Associate Professor with the Department of Electronics and Communication Engineering, University of Illinois at Urbana-Champaign, Champaign, IL, USA, since 2011. His current research interests include system-level and high-level synthesis, nano-systems design and nano-centric CAD techniques, graphics processing unit and reconfigurable computing, hardware/software co-design, and computational biology.

Dr. Chen was a recipient of the NSF CAREER Award in 2008, five Best Paper Awards from ASPDAC'09, SASP'09, FCCM'11, SAAHPC'11, and CODES+ISSS'13, ACM SIGDA Outstanding New Faculty Award in 2010, and the IBM Faculty Award in 2014 and 2015. He is or has been an Associated Editor for TCAD, TODAES, TVLSI, and TCAS-I.