# An Efficient Compiler Framework for Cache Bypassing on GPUs

Xiaolong Xie[1], Yun Liang[1]* , Guangyu Sun[1] and Deming Chen[2]
[1]Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, China
[2]University of Illinois, Urbana-Champaign, USA
{xiexl_pku,ericlyun,gsun}@pku.edu.cn,{dchen}@illinois.edu

## 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 the 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 integrate our techniques into an automatic compiler framework that leverages PTX instruction set architecture. Experiments evaluation demonstrates that compared to cache-all and bypass-all solutions, our techniques can achieve considerable performance improvement.

## Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures; D.3.4 [**Processors**]: Compilers

## General Terms

Performance, Measurement, Algorithms

## Keywords

GPU, Cache Bypassing, Compiler Optimization

*Corresponding Author

## 1. INTRODUCTION

With the continuing evolution of heterogenous computing platforms that consist of 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, 16, 28]. 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 [9], Qualcomm's Snapdragon series with Adreno GPU [10], and Samsung's Exynos series with ARM Mali GPU [11].

Despite the high computing power of GPUs, performance optimization of GPUs is challenging [34]. The achieved performance speedup critically depends on memory subsystem [19, 25]. In early GPUs, software-managed scratchpad memory (SPM) 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 effectively improve the memory performance. For example, both NVIDIA Fermi and Kepler architectures feature configurable L1 cache [3, 4] in conjunction with scratchpad memory (a.k.a shared memory); they also introduce a unified L2 cache to further exploit data reuse.

For both CPUs and GPUs, cache can effectively hide the data access latency by exploiting the temporal and spatial localities of a program. 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 the compiler to explicitly control the L1 cache access or bypass for global load instructions [5]. Cache bypassing is very beneficial for applications with memory accesses that are scattered or have no data reuse as it can help to improve memory efficiency and reduce cache pollution [14].

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 can not be considered 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 [23]. 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.

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 among them, load efficiency, and memory traffic. 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. Our framework leverages the Parallel Thread Execution (PTX) instruction set architecture (ISA). Experimental results show that our compiler framework for cache bypassing can effectively optimize the overall GPU performance.

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

- 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.

- 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 and conflicts between global load instructions.

- Evaluation. Experiments on a variety of applications show that compared to cache-all solution, our techniques improve the average cache benefits from 4.4% to 12.9% for 16 KB L1 cache. The performance speedup of our cache bypassing techniques is up to 2.62X.

This paper is organized as follows. In section 2, we provide some background on GPUs and present a motivational example for our cache bypassing study. In section 3, we introduce our compiler framework and the involved analysis components. In section 4 and section 5, we detail the characterization and optimization components. Section 6 presents the experimental results. Section 7 discusses the related work and section 8 concludes the paper.

## 2. BACKGROUND AND MOTIVATION

This section provides the background details and motivation of our study. We use NVIDIA Kepler GTX 680 GPU architecture and CUDA terminology [6] in this paper. But our techniques are equally applicable to other GPUs with caches and the OpenCL programming models.

### 2.1 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 GTX 680 used in this paper has 8 SMs, each of which has 192 SPs. Thus, there are 1536 cores in total. Each SM has private registers which

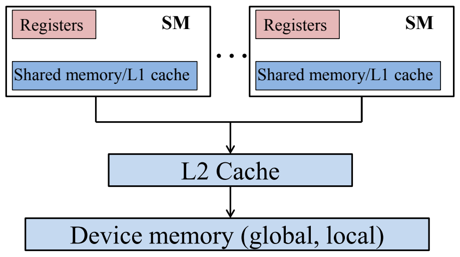

**Figure 1: GPU Memory Hierarchy**

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. On one SM threads are scheduled in units of warps (32 threads). The threads in a warp execute in a SIMD style. GPUs perform zero overhead scheduling to interleave warps on a SM to hide memory access latency and pipeline stall.

Figure 1 shows the memory hierarchy of recent generations of GPUs with caches. Each SM is equipped with caches in conjunction with shared memory. For example, each SM of NVIDIA Fermi and Kepler architectures contains a configurable 64 KB on-chip memory which is shared by scratchpad memory and L1 data cache [3, 4]. The programmer can choose how much storage to devote to the L1 cache versus scratchpad memory (16 vs 48, 32 vs 32, 48 vs 16). All the SMs share a unified L2 cache. L1 cache has 128 bytes block size while L2 cache has 32 bytes block size. The block size does not change with the partition between L1 cache and scratchpad memory. 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. Most GPU applications begin data accesses from global memory and write results back to global memory. Local memory is used as a per-thread private memory space for register spills, function calls, and automatic array variables. Hence, 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 1 describes the architecture details of NVIDIA GTX 680 (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-grained 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 (see section 3 for detailed program interface). 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 cache, 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.

### 2.2 Motivation

Here, we motivate the performance speedup potential through cache bypassing on GPUs and the need of automatic compiler framework for cache bypassing. We use the kernel *particle filter* from Rodinia benchmark suite [18] as a case study. There are totally 14 load instructions in *particle filter*. Due to the large design space ($2^{14}$), we only choose 6 global load instructions with high access frequencies to be cache bypassing candidates. For the rest 8 global

**Table 1: 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 |

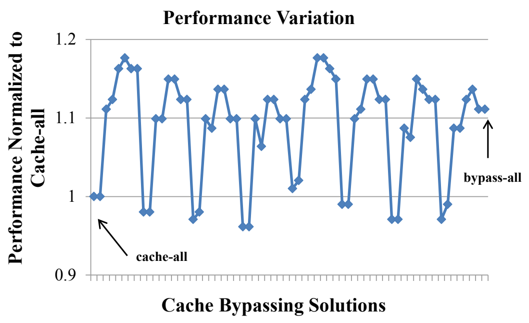

**Figure 2: Performance variation of *particle filter* with different cache bypassing solutions.**

load instructions, we choose to cache all of them. Figure 2 shows how the performance speedup varies with the cache bypassing solutions and the results are normalized to cache-all (e.g. cache all the 6 chosen global load instructions). The horizontal axis represents the subset of design space ($2^6 = 64$ solutions) we consider, the left-most point represents cache-all and the right-most point represents bypass-all. The experiments are performed on NVIDIA Kepler GTX 680 and we use NVIDIA profiler [7] to collect the performance data.

As shown in Figure 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.

## 3. COMPILER FRAMEWORK

Figure 3 presents our compiler framework for cache bypassing on GPUs. The CUDA code is first pre-compiled to PTX (Parallel Thread Execution) code which is CUDA's intermediate representation used in NVIDIA CUDA Compiler [6]. Our compiler framework takes unmodified PTX code as input and outputs optimized CUDA binaries. The compiler framework involves three components: characterization, optimization and instrumentation. Initially, characterization component collects the data access, reuse, load efficiency, and memory traffic 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 PTX ISA. The characterization component is also assisted with the instrumentation component for profiling.

The characterization and optimization components are detailed in sections 4 and 5, respectively. Here, we describe the details of the instrumentation component. Our implementation leverages the PTX ISA [5]. Global memory loads in PTX are in the following format,

$$\text{ld.global.}l1\_cache\_option.type\ dst, src$$

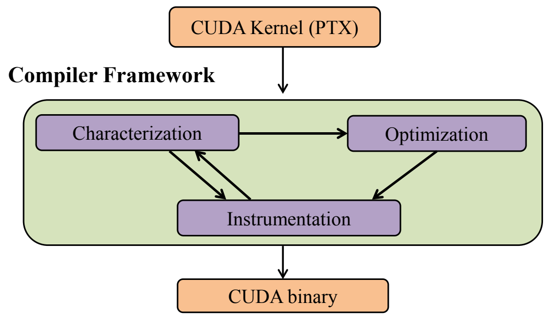

**Figure 3: Compiler framework for cache bypassing on GPUs.**

*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 the default setting.

We can explicitly control cache access or bypass by modifying the PTX code. More clearly, we implement cache access using cache option $ca$ and implement cache bypass using cache option $cg$ for the global load instruction. After all these changes, we need to update the PTX section size and embed it into CUDA binary. Below, it gives an example about PTX code instrumentation. After PTX code instrumentation, the first global load instruction (line 1) bypasses the L1 cache while the last global load instruction (line 4) accesses the L1 cache.

```
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**

## 4. 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 used for cache bypassing optimization (section 5).

### 4.1 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 NVIDIA profiler [7], 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 $\frac{16}{128} = 12.5\%$ and $E_{off}$ is $\frac{16}{32} = 50\%$. $E_{on}$ ($E_{off}$) can be collected using CUDA profiler [7]. Hence, bypassing L1 cache 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, bypassing L1 cache can help to improve the load efficiency and thus reduce memory traffic.

## 4.2 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 threads in a thread block execute together in lock-step style, then it is less likely for caches to exploit temporal locality. Indeed, previous work [23] makes such a pessimistic assumption. 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.

- Intra-warp spatial/temporal locality: the threads within the same warp access the same cache line.
- Inter-warp spatial/temporal locality: the threads within different warps access the same cache line.

The threads in a warp are executed in a SIMD style. Intra-warp spatial locality refers to the memory coalescing for GPUs [6]. The continuous memory accesses from the threads within a warp lead to coalesced accesses. Intra-warp spatial locality is determined by the data access pattern of the warp and the coalescing rules of the GPU architecture. Cache bypassing has no effect on it. Intra-warp temporal locality is important 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, intra-warp 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. On the other hand, inter-warp locality is also possible because different warps may access the same data (e.g. array [tid % 32]). Cache bypassing is very beneficial for applications without data localities because it helps to decrease the cache conflicts.

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

## 4.3 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 $i^{th}$ global load instruction. We rely on the instrumentation component described in section 3 to modify the GPU code and use NVIDIA profiler [7] to collect the following metrics,

- $access_i$: the number of L1 cache accesses for $ld_i$. This number is obtained through profiler by bypassing L1 cache for all the global load instructions except $ld_i$.
- $hit_i$: the number of L1 cache hits for $ld_i$. This number is obtained as a by-product of $access_i$.
- $hit_{i,j}$: the number of L1 cache hits for $ld_i$ and $ld_j$ together. This number is obtained through profiler by bypassing L1 cache for all the global load instructions except $ld_i$ and $ld_j$.

Let us define $gain_{i,j}$

$$gain_{i,j} = hit_{i,j} - (hit_i + hit_j)$$

We use $gain_{i,j}$ to measure the data reuses or conflicts between $ld_i$ and $ld_j$. Note that $gain_{i,j}$ may be either positive or negative. If $gain_{i,j}$ 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 $gain_{i,j}$ to estimate L1 cache hit ratio. However, L1 cache hit ratio does not predict performance well as demonstrated in prior work [23]. High L1 hit ratio 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 $T_{on}(ld_i)$ ($T_{off}(ld_i)$) to denote the L2 cache traffic when L1 cache is accessed (bypassed) for $ld_i$.

$$T_{on}(ld_i) = (access_i - hit_i) \times L1\_block\_size$$

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 $4 = \frac{128}{32}$. 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, we can not get the exact number of transferred L2 blocks for each global load instruction through NVIDIA profiler [7] as the L2 cache can not be bypassed. Instead, we compute $T_{off}(ld_i)$ as follows,

$$T_{off}(ld_i) = \frac{access_i \times L1\_block\_size \times E_{on}}{E_{off}}$$

where $E_{on}$ ($E_{off}$) is the program global load efficiency when L1 cache is accessed (bypassed). For the computation of $T_{off}(ld_i)$, we use the overall load efficiency for all instructions in a program.

DEFINITION 1 (**Traffic Reduction Graph**). *Let traffic reduction graph $TG = (V, E)$ 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 $v_i$, $W(v_i) = T_{off}(ld_i) - T_{on}(ld_i)$; the weight of edge $e(v_i, v_j)$, $W(e(v_i, v_j)) = gain_{i,j} \times L1\_block\_size$.*

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 experiment section). 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. For this paper, we use the same input for profiling and evaluation.

## 5. OPTIMIZATION COMPONENT

### 5.1 Problem Formulation

Given $N$ global load instructions, we could select a subset of global load instructions for cache bypassing. Thus, there exists $2^N$ 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 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 5.1. *Traffic Reduction Maximization Problem is NP-Hard.*

PROOF. We show a reduction from Maximal Clique problem [12]. 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 $\forall e \in E' \setminus E, W(e) = -\infty$. 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. □

### 5.2 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 loads.

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

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

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.

$$N_{v_i}, M_{v_i,v_j} = 0 \text{ or } 1$$
$$M_{v_i,v_j} \le N_{v_i}$$
$$M_{v_i,v_j} \le N_{v_j}$$
$$M_{v_i,v_j} \ge N_{v_i} + N_{v_j} - 1$$

**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           $T_{other}(v_i) = \sum_{v_j \in V_{cache}} W(e(v_i, v_j)) + \sum_{v_k \in V_{remain} \wedge v_k \neq v_i} W(e(v_i, v_k))$
7           if $T_{other}(v_i) \le min\_T$ then
8               $min\_T = T_{other}(v_i)$; $min\_v = v_i$;
9
10      $T = T_{other}(min\_v) + W(min\_v)$ ;
11      if ($T \le 0$) then
12          //bypass it;
13          delete ($min\_v$) from $TG$;
14          add $min\_v$ to $V_{bypass}$;
15      else
16          //cache it;
17          add $min\_v$ to $V_{cache}$;
18      delete ($min\_v$) from $V_{remain}$ ;
19
```

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

### 5.3 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 instruction, we first evaluate its potential traffic reduction if it is cached together with other cached and remaining global loads (line 6). We selects 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 traffic reduction graph; otherwise it is kept in the traffic reduction graph for evaluation of the remaining nodes.

Figure 4 shows an example of our algorithm. The traffic reduction graph consists of four nodes (four global loads). 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 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.

## 6. EXPERIMENTS

**Experiments Setup**. We evaluate our techniques on NVIDIA GTX 680 (Kepler Architecture). The hardware details of GTX 680

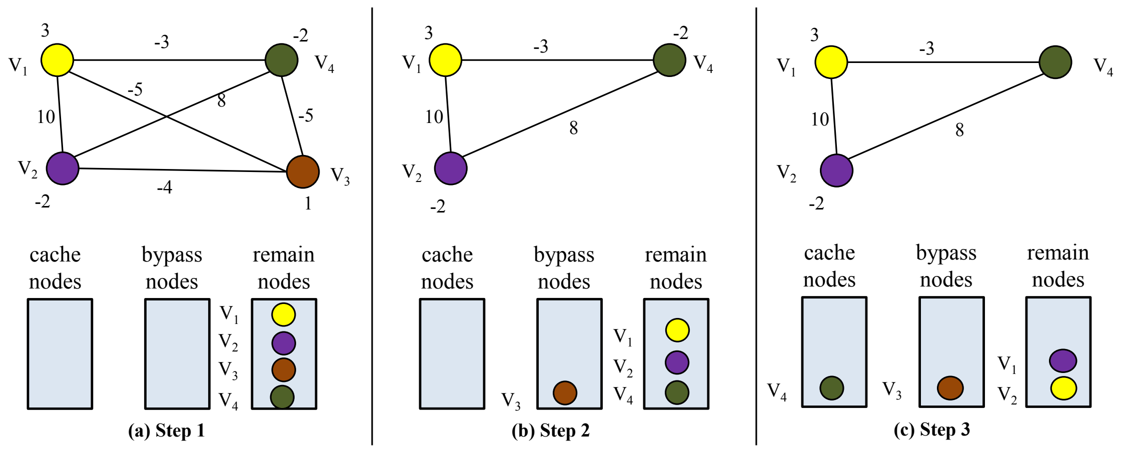

**Figure 4: Illustration of our heuristic algorithm.**

**Table 2: Benchmark Characteristics.**

| Benchmark | Source | Number of Thread Blocks | Thread Block Size | Shared Memory (KB) |
|---|---|---|---|---|
| backprop (BAA) | Rodinia [18] | 16384 | 256 | 0 |
| bfs (BFS) | Rodinia [18] | 1954 | 512 | 0 |
| euler3d (EUF) | Rodinia [18] | 1212 | 192 | 0 |
| kmeans (KMI) | Rodinia [18] | 3249 | 256 | 0 |
| particle filter (PFFL) | Rodinia [18] | 2 | 512 | 4 |
| srad prepare (SRP) | Rodinia [18] | 450 | 512 | 0 |
| srad reduce (SRR) | Rodinia [18] | 1 | 512 | 4 |
| srad kernel (SRS) | Rodinia [18] | 450 | 512 | 0 |
| spmv (SPM) | Parboil [15] | 765 | 192 | 0 |
| mri-gridding (MGR) | Parboil [15] | 5188 | 512 | 0 |
| mri-q (MRQ) | SDK [8] | 4 | 32 | 0 |

are presented in Table 1. We select a set of benchmarks from benchmark suite Rodinia [18], Parboil [15], and NVIDIA GPU Computing SDK [8]. The tested benchmarks are general-purpose GPU applications with diverse characteristics including thread structures, computation, and memory access patterns. Some of them are memory intensive applications that involve a large number of global loads and stores while the rest are computation intensive application that contain only a few number of global loads and stores. The benchmark details are shown in Table 2.

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. We implement the optimal solution based on the ILP and heuristic algorithms. We use MOSEK [2] to solve the ILP problem. NVIDIA GTX 680 has configurable L1 cache. It can be configured to 16, 32, and 48 KB. Thus, we evaluate our technique using three different cache sizes.[1] The performance are measured through NVIDIA Profiler [7].

**Performance Speedup**. For each benchmark, we compare four solutions: bypass-all, cache-all, Heuristic, and ILP. For cache-all solution, all the global load instructions go through L1 cache; for bypass-all solution, all the global load instruction bypass L1 cache. Figure 5 presents the results for three different cache sizes. We normalize the performance to bypass-all solution.

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 with only a small number of loads, but most likely give bad performance for benchmarks with large number of loads. In contrast, our heuristic solution performs consistently well across all the benchmarks. The performance speedup of our cache bypassing techniques is up to 2.62X. More clearly, for 16 KB cache, our heuristic improves the performance by 12.9% on average while cache-all improves the performance by only 4.4% on average. For 32 KB cache, our heuristic improves the performance by 17.1% on average while cache-all improves the performance by 10.6% on average. For 48 KB cache, our heuristic improves the performance by 21.4% on average while cache-all improves the performance by 18.3% on average. The average value is computed using geometric mean.

The performance improvement of our heuristic linearly increases as the cache size increases. This is because larger cache offers more opportunities to exploit data localities than smaller cache. Cache size increases at the cost of the shared memory decrease. The decrease of shared memory does not affect the performance much for most of the benchmarks as these benchmarks either do not use or just use a small portion of the shared memory as shown in Table 2. We also notice that the gap between cache-all and our approach decreases as the cache size increases, that is because more global loads can fit into the larger caches.

Our heuristic and ILP solution return the same results for most of the benchmarks. However, there are a few cases that our heuristic is slightly better than ILP solution. This is because for those benchmarks the load efficiency of different loads are diverse and thus using a uniform load efficiency is not accurate. Therefore, it is possible that our ILP solution results in a sub-optimal solution in practice. But overall our heuristic and ILP solution perform consistently well across all the benchmarks and cache settings.

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

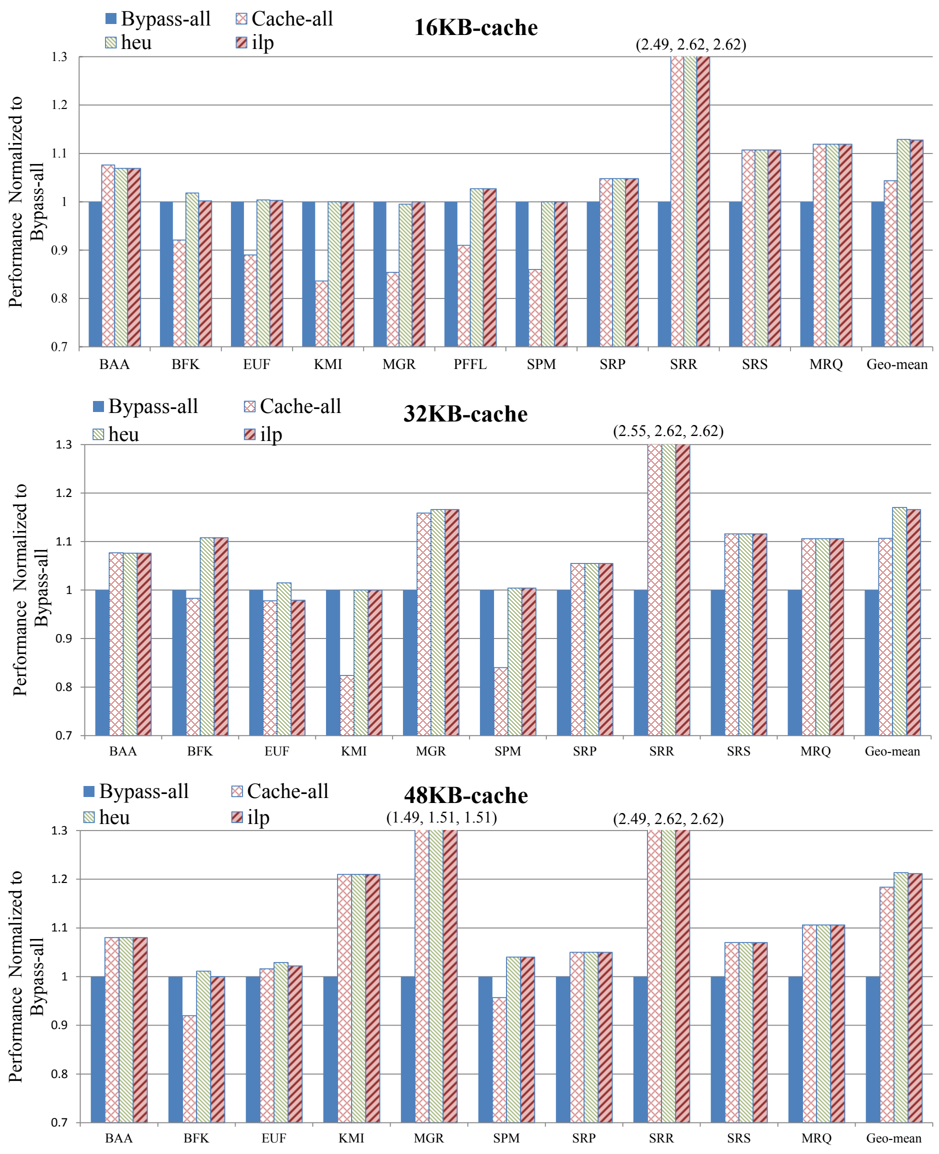

**Figure 5: Performance comparison for different cache sizes.**

Prior cache bypassing technique [23] does not model cross instruction localities. However, different global instructions may access the same cache line and a warp may execute multiple instructions before switching to the next warp. Hence, intra- and inter-warp temporal localities are important for GPU architecture. Our technique achieves high improvement as the traffic reduction graph captures these localities. For example, for MRQ benchmark, our solution achieves 11.9%, 10.6% and 10.6% improvement for 16, 32 and 48 KB cache compared to bypass-all solution, respectively. However, for MRQ benchmark, prior work [23] chooses bypass-all solution for all three cache sizes due to the neglect of cross instruction localities.

**Efficiency**. Our compiler framework runs very efficiently. For all the benchmarks, it only take a few seconds to complete.

## 7. RELATED WORK

**GPU Performance Optimization**. Although GPUs promised high performance, tuning GPUs for high performance was not a trivial task [19]. Both analytical performance models and optimization techniques had been developed [22, 25]. The state-of-the-art of GPU performance optimization techniques focused on automatic data movement, data layout transformation, thread and warp scheduling, control flow divergence elimination, register allocation optimization, and memory coalescing optimization [17, 27, 31, 34, 35, 13]. However, none of above works targeted GPU architecture with caches.

There were very few studies on GPU caches. Recently, Jia et al. presented a characterization and optimization study for GPU caches [23]. Their characterization study demonstrated that on GPUs L1 cache hit ratio does not correlate with performance. They used static analysis to analyze data access patterns. However, there was one major drawback in their work. They assumed there was no data reuse between global load instructions and different iterations of the same global load instructions. Thus, their method only considered inter-warp spatial locality and completely neglected other localities. In contrast, our solution systematically captured both temporal and spatial localities using traffic reduction graph. Kuo et al. presented a cache capacity aware thread scheduling for irregular memory access on GPUs [26]. However, they did not explore cache bypassing in their work.

**Cache Bypassing for CPUs**. Cache bypassing had been widely used for CPU caches to effectively alleviate the cache pressure. Runtime cache bypassing with extra hardware supports had been used to reduce cache pollution in [30, 24]. As an alternative to hardware approach, compiler-assisted cache bypassing techniques had been proposed too [20, 33]. They used hit ratio as performance metrics to guide the cache bypassing. These techniques were not applicable to GPUs as hit ratio did not correlate well with performance on GPUs as shown in [23].

**Memory Customization for Embedded System**. The customization of memory subsystem is critical for embedded systems. Hardware and software techniques had been proposed for embedded systems with scratchpad memory and cache [32, 21, 29]. They mainly used hit ratio as performance metrics and thus were not applicable to GPUs.

## 8. 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 load instruction. This feature benefits the general purpose applications with scattered dynamic data access patterns. 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 conflicts with others. Experiments using a set of real applications show that our techniques improve the average cache benefits to 12.9%, 17.1%, and 21.4% for 16, 32, and 48 KB caches, respectively.

## 9. ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China (No. 61202072) and SRC/DARPA CFAR center.

## 10. REFERENCES

[1] GE Intelligent Platforms. `http://defense.ge-ip.com/products/hpec/c560`.

[2] Mosek. `http://www.mosek.com/`.

[3] NVIDIA. Fermi GPUs `www.nvidia.com/object/fermi-architecture.html`.

[4] NVIDIA. Kepler GPUs `www.nvidia.com/object/nvidia-kepler.html`.

[5] NVIDIA. PTX Code `http://docs.nvidia.com/cuda/pdf/ptx_isa_3.1.pdf`.

[6] NVIDIA. CUDA Programming Guide, Version 3.2.

[7] NVIDIA. Profiler `http://docs.nvidia.com/cuda/profiler-users-guide/index.html`.

[8] NVIDIA GPU Computing SDK. `http://developer.nvidia.com/gpu-computing-sdk`.

[9] NVIDIA Tegra. `http://www.nvidia.com/object/tegra.html`.

[10] Qualcomm Inc. `http://www.qualcomm.com/snapdragon`.

[11] SamSung Inc. `www.samsung.com/exynos`.

[12] T. Cormen, C. Stein, R. Rivest, and C. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[13] Z. Cui, Y. Liang, K. Rupnow, and D. Chen. An accurate GPU performance model for effective control flow divergence optimization. In *IPDPS*, 2012.

[14] C. J. Wu et al. SHiP: signature-based hit predictor for high performance caching. In *Micro*, 2011.

[15] J. A. Stratton et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. In *IMPACT Technical Report*, 2012.

[16] J. D. Owens et al. GPU computing. *Proceedings of the IEEE*, 2008.

[17] M. M. Baskaran et al. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS*, 2008.

[18] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[19] S. Ryoo et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, 2008.

[20] Y. Wu et al. Compiler managed micro-cache bypassing for high performance EPIC processors. In *Micro*, 2002.

[21] P. Francesco et al. An integrated hardware/software approach for run-time scratchpad management. In *DAC*, 2004.

[22] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.

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

[24] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. In *IEEE Transactions on Computers*, 2008.

[25] Y. Kim and A. Shrivastava. CuMAPz: A tool to analyze memory access patterns in CUDA. In *DAC*, 2011.

[26] H. Kuo, T. Yen, B. C. Lai, and J. Jou. Cache capacity aware thread scheduling for irregular memory access on many-core GPGPUs. In *ASPDAC*, 2013.

[27] Y. Liang, Z. Cui, K. Rupnow, and D. Chen. Register and thread structure optimization for GPUs. In *ASPDAC*, 2013.

[28] Y. Liang et al. Real-time implementation and performance optimization of 3D sound localization on GPUs. In *DATE*, 2012.

[29] Y. Liang and T. Mitra. Static analysis for fast and accurate design space exploration of caches. In *CODES+ISSS*, 2008.

[30] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache Bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Micro*, 2008.

[31] IJ. Sung, J. A. Stratton, and W. W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *PACT*, 2010.

[32] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, May 2006.

[33] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *PACT*, 2002.

[34] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *PLDI*, 2010.

[35] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*, 2011.