Optimizing Data Placement on GPU Memory: A Portable Approach

Guoyang Chen, Xipeng Shen, *Senior Member, IEEE*, Bo Wu, and Dong Li, *Member, IEEE*

Abstract—Modern GPUs feature complex memory system designs. One GPU may contain many types of memory of different properties. The best way to place data in memory is sensitive to many factors (e.g., program inputs, architectures), making portable optimizations of GPU data placement a difficult challenge. PORPLE is a recently proposed method that overcomes the difficulties by enabling online optimizations of data placement through a three-way synergy: a specification language for memory system description, a compiler framework for data access analysis and code staging, and a runtime library for efficiently finding and materializing data placement on the fly. This article provides a comprehensive description of this method, and presents several extensions that significantly improve the scalability of PORPLE, which include a novel algorithm design for efficiently searching for the best data placements, the use of active profiling for reducing the online-profiling overhead, and a systematic examination of a path-based performance model. By automatically tailoring data placements for each execution of a GPU program, the enhanced PORPLE brings significant speedups (1.72X on average) to many GPU kernels across GPU architectures and program inputs.

Index Terms—GPU, memory performance, cache, compiler, data placement, hardware specification language

1 Introduction

With its massive parallelism and tremendous computing power, Graphic Processing Units (GPU) have received a rapid adoption in modern computing. To meet the ever growing demands for data by the large number of computing units, modern memory systems become complex, sophisticated, and heterogeneous. One memory system often consists of a number of components, which each have some different properties (e.g., access bandwidth, access latency, endurance, memory organization, preferable access patterns, and programming paradigms). For example, on an NVIDIA Kepler GPU, there are more than eight types of memories (global, texture, shared, constant, and various caches), with some on-chip, some off-chip, some directly manageable by software, and some not. Even for a single type of memory, there are often several ways to access it with different performance implications (e.g., some going through cache, some not, some enjoying 2-D locality, some not). Emerging memory technologies (e.g., 3D stacked memory, non-volatile memory) may further increase the variety of memory components in a future memory system.

The sophisticated design of memory systems is a double-edged sword. Studies have shown that finding the suitable kinds (or pieces) of memory to place data—called the *data placement problem*—can significantly improve, sometimes double, the performance of some already manually optimized programs [1], [2], [3]. On the other hand, the complexity and fast evolvement of such memory systems make the potential difficult to tap into. Recent studies show that many memory-intensive GPU applications carefully written by developers cannot yet reach half of the performance that they could achieve with a better memory usage [1], [2]. Yet, manual efforts have been what existing programming systems (e.g., CUDA, OpenCL) all require. The urgency for addressing the issue is likely to increase even more as emerging memory technologies (e.g., Non-volatile memory, 3D stacked memory) add more kinds of components and complexities into future memory systems.

Prior efforts for solving the problem have concentrated on two directions. The first is offline auto-tuning, which tries many different placements and measures the performance on some training runs [4]. This approach is time-consuming, and cannot easily adapt to the changes in program inputs or memory systems. The second is to use some high-level rules derived empirically from many executions on a GPU [1]. These rules are straightforward, but as shown later in this paper, they often fail to produce suitable placements, and their effectiveness degrades further when GPU memory systems evolve across generations of GPU.

This paper presents PORPLE, a portable data placement engine that enables a portable and extensible way to solve the data placement problem. A key feature of PORPLE is its *strategy of lazy placement*: It finds and realizes the appropriate data placement during the execution time of the program in question; when necessary, it postpones the recognition of data access patterns of the program to the execution time as well. The laziness makes it possible to recognize the input-sensitive attributes of data accesses (e.g., patterns, sizes) and finds the placements suiting the current execution by matching up those attributes with the properties of underlying

- *G. Chen and X. Shen are with North Carolina State University, Raleigh, NC 27695. E-mail: {gchen11, xshen5}@ncsu.edu.*
- *B. Wu is with the Colorado Schools of Mines, Golden, CO 80401. E-mail: bwu@mines.edu.*
- *D. Li is with the University of California, Merced, CA 95340. E-mail: dli35@ucmerced.edu.*

Manuscript received 25 Feb. 2016; revised 30 June 2016; accepted 1 Aug. 2016. Date of publication 29 Aug. 2016; date of current version 16 Feb. 2017. Recommended for acceptance by R. F. DeMara.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2016.2604372

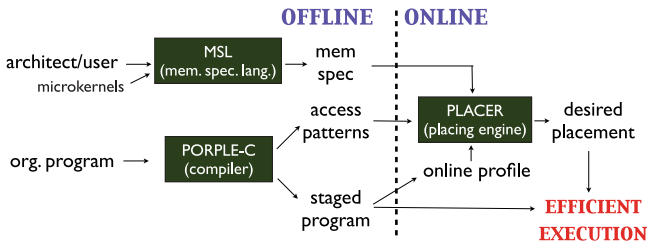


Fig. 1. High-level structure of PORPLE.

architecture. The unique design gives PORPLE some important properties, making its data placement automatically adapt to both program input changes and architectural differences. It also grants PORPLE a broad applicability, making it able to handle both regular (e.g., with affine memory accesses) and irregular programs (e.g., with indirect memory accesses).

The appealing properties come from the support offered by a set of techniques. As shown in Fig. 1, at a high level, PORPLE contains three key components: a specification language MSL for providing memory specifications, a compiler PORPLE-C for revealing the data access patterns of the program and staging the code for runtime adaptation, and an online data placement engine *Placer* that consumes the specifications and access patterns to find the best data placements at runtime.

Memory specification language (MSL), is a carefully designed small specification language. It provides a simple, uniform way to specify a type of memory and its relations with other pieces of memory in a system. It features a "serialization condition" construct that allows a systematic specification of all the special properties of the various types of GPU memory. With this design, extending the language coverage to include a new memory system can be achieved by simply adding a new entry into the MSL specification.

PORPLE-C is a source-to-source compiler, which analyzes program data access patterns, and transforms a GPU program into a *placement-agnostic form* such that it can work with an arbitrary data placement decided by the data placement engine at runtime. The form is equipped with some guarding statements. Executions of the program can automatically select the appropriate version of code to access data according to the current data placement.

Placer is the runtime engine of PORPLE, embodied by a set of library functions inserted into the target program by PORPLE-C. It features a lightweight bandwidth-centric performance model, useful for efficiently assess the profits of candidate placements. It includes a hybrid, extensible search module to support effective and scalable search for appropriate data placements on the fly. It uses online profiling to find out the data access patterns of irregular programs, and employs active learning to minimize the overhead.

Together, these techniques make PORPLE an extensible data placement engine, offering a general solution to GPU data placement. It adapts to inputs and memory systems; it allows easy extension to new memory systems; it requires no offline training; in most cases, it optimizes data placement transparently (i.e., with no need for manual code modification). In exceptional cases where automatic code transformation is difficult to do, it is still able to offer suggestions for code refactoring. Our experiments on three generations of GPU show that POPRLE successfully finds optimal or near-optimal data placement across inputs and

Keywords:
address1, address2, index1, index 2, banks, blockSize, warp, block, grid, sm, core, tpc, die, clk, ns, ms, sec, na, om, ?;
// na: not applicable; om: omitted; ?: unknown; // om and ? can be used in all fields

Operators:
C-like arithmetic and relational operators, and a scope operator {};

Syntax:
- specList ::= processorSpec memSpec*
- processorSpec ::= die=Integer tpc; tpc=Integer sm; sm=Integer core; end-of-line
- memSpec ::= name id swmng rw dim size blockSize banks latency upperLevels lowerLevels shareScope concurrencyFactor serialCondition ; end-of-line
- name ::= String
- id ::= Integer
- swmng ::= Y | N *// software manageable or not*
- rw ::= R|W|RW *// allow read or write accesses*
- dim ::= na | Integer *// special for arrays of a particular dimensionality*
- sz ::= Integer[K|M|G|T|ε][E|B] *// E for data elements*
- size ::= sz | <sz sz> | <sz sz sz>
- blockSize ::= sz | <sz sz> | <sz sz sz>
- lat ::= Integer[clk|ns|ms|sec] *// clk for clocks*
- latency ::= lat | <lat lat>
- upperLevels ::= <[id | name]*>
- lowerLevels ::= <id*>
- shareScope ::= core | sm | tpc | die
- concurrencyFactor ::= < Number Number>
- serialCondition ::= scope{RelationalExpr}
- scope ::= warp | block | grid

Fig. 2. Syntax of MSL with some token rules omitted.

architectures, yielding up to 2.63X (1.72X on average) speedups on a set of regular and irregular GPU benchmarks, outperforming a rule-based method [1] significantly.

Some features of PORPLE have been described in two earlier papers [2], [3]. This article provides a more complete description, including three new extensions that significantly improve the scalability and efficiency of PORPLE: (1) It replaces the previous memory latency-centric performance model with a path-based performance model (Section 3.1). Even though the new model was briefly mentioned in a 5-page magazine version [3], this paper gives the first comprehensive description and evaluation. (2) It complements the branch-and-bound search algorithm with a greedy algorithm to make the search scalable (Section 3.2). (3) It introduces active learning into the online profiling process to significantly reduce the profiling overhead (Section 3.3). The evaluation section (Section 5) presents the results after the framework gets extended with the three techniques, and added a study on the scalability of the technique (Section 5.3.4) on two kernels from a real-world large application containing many arrays.

In the following, we present each of the major components of PORPLE and then report the experimental results.

2 MSL: Specification for Extensibility

An important feature of PORPLE is that it is easy to extend to new memory systems. We achieve this feature by MSL.

MSL is a small language designed to provide an interface for compilers to understand a memory system. Fig. 2 shows its keywords, operators, and syntax written in Backus–Naur Form (BNF). An MSL specification contains one entry for processor and a list of entries for memory. We call each entry a *spec*. The processor entry shows the composition of a die, and an SM.

Each memory spec corresponds to one type of memory, indicating the name of the memory (started with letters)

```
Examples of serialization conditions:
 constant mem:
    warp{address1 != address2}
 shared mem:
    block{word1!=word2 && word1%banks == word2%banks}
 global mem:
    warp{⌊address1/blockSize⌋ != ⌊address2/blockSize⌋}
```

Fig. 3. Example serialization conditions in MSL.

and a unique ID (in numbers) for the memory. The name and ID could be used interchangeably; having both is for conveniences. The field "swmng" is for indicating whether the memory is software manageable. The data placement engine can explicitly put a data array onto a software manageable memory (versus hardware managed cache for instance). The field "rw" indicates whether a GPU kernel can read or write the memory. The field "dim" is designed for memory that manifest different properties when the array has a different dimension (e.g., texture memory). If it is not "?", the spec entry is applicable only when the array dimensionality equals to the value of "dim". The field after "dim" indicates memory size. Because a GPU memory typically consists of a number of equal-sized blocks or banks, "blockSize" (which could be multi-dimensional) and the number of banks are next two fields in a spec. The next field afterwards describes memory access latency. To accommodate access latency difference between read and write operations, the spec allows the use of a tuple to indicate both. We use "upperLevels" and "lowerLevels" to indicate memory hierarchy; they contain the names or IDs of the memories that sit above (i.e., closer to computing units) or below the memory of interest. The "shareScope" field indicates in what scope the memory is shared. For instance, "sm" means that a piece of the memory is shared by all cores on a streaming multiprocessor.

The "concurrencyFactor" is a field that indicates parallel transactions a memory (e.g., global memory and texture memory) may support for a GPU kernel. Its inverse is the average number of memory transactions that are serviced concurrently for a GPU kernel. As shown in previous studies [5], such a factor depends on not only memory organization and architecture, but also kernel characterization. MSL broadly characterizes GPU kernels into compute-intensive and memory-intensive, and allows the "concurrencyFactor" field to be a tuple containing two elements, respectively corresponding to the values for memory-intensive and compute-intensive kernels. We provide more explanation of "concurrencyFactor" through an example later in this section, and explain how it is used in the next section.

GPU memories often have some special properties. For instance, shared memory has an important feature called bank conflict: When two accesses to the same bank of shared memory happen, they have to be served serially. But on the other hand, for global memory, two accesses by the same warp could be coalesced into one memory transaction if their target memory addresses belong to the same segment. While for texture memory, accesses can benefit from 2-D locality, constant memory has a much stricter requirement: The accesses must be to the same address, otherwise, they have to be fetched one after another.

How to allow a simple expression of all these various properties is a challenge for the design of MSL. We address it based on an insight that all these special constraints are essentially about the conditions for multiple concurrent accesses to a memory to get serialized. Accordingly, MSL introduces a field "serialCondition" that allows the usage of simple logical expressions to express all those special properties. Fig. 3 shows example expressions for some types of GPU memory. Such an expression must start with a keyword indicating whether the condition is about two accesses by threads in a warp or a thread block or a grid, which is followed with a relational expression on the two addresses. It also uses some keywords to represent data accessed by two threads: *index1* and *index2* stand for two indices of elements in an array, *address1* and *address2* for addresses, and *word1* and *word2* for the starting addresses of the corresponding words (by default, a word is 4-byte long). For instance, the expression for shared memory, "block{word1≠word2 && word1%banks==word2%banks}", claims that when the words accessed by two threads in the same thread block are different and fall onto the same bank (which is a bank conflict), the two accesses get serialized. The expression for constant memory claims that if two threads in a warp access the same address, one memory transaction is enough (because of the broadcasting mechanism of constant memory); they however get serialized otherwise. This simple way of expression makes it possible for other components of PORPLE to easily leverage the features of the various memory to find good data placements as next section shows.

Fig. 4 shows the exerted MSL specification for Tesla M2075. Three special notations "?", "om", and "na" are used for unknown, inferrable, or not applicable info. For instance, L2 has "om" in its upperLevels field as it is inferrable from

```
Mem spec of Tesla M2075:

 die =1 tpc; tpc = 16 sm; sm = 32 core;
 globalMem 8 Y rw na 5375M 128B ? 600clk <L2 L1> <> die <0.1 0.5> warp{⌊address1/blockSize⌋ != ⌊address2/blockSize⌋};
 L1 9 N rw na 16K 128B ? 80clk <> <L2 globalMem> sm ? warp{⌊address1/blockSize⌋ != ⌊address2/blockSize⌋};
 L2  7 N rw na  768K 32B ? 390clk om om die ? warp{⌊address1/blockSize⌋ != ⌊address2/blockSize⌋};

 constantMem 1 Y r na 64K ? ? 360clk <cL2 cL1> <> die ? warp{address1 != address2};
 cL1 3 N r na 4K 64B ? 48clk <> <cL2 constantMem> sm ? warp{⌊address1/blockSize⌋ != ⌊address2/blockSize⌋};
 cL2 2 N r na 32K 256B ? 140clk <cL1> <cL2 constantMem> die ? warp{⌊address1/blockSize⌋ != ⌊address2/blockSize⌋};

 sharedMem 4 Y rw na 48K ? 32 48clk <> <>  sm ? block{word1!=word2 && word1%banks ==word2%banks};

  ...   ...
```

Fig. 4. Memory specification of Tesla M2075 in MSL ("?": unknown; "om": ommitted, inferrable from other entries; "na": not applicable).

other spec entries. In this example, the concurrency factors of global and texture memory are set to 0.1 for memory-intensive GPU kernels and 0.5 for compute-intensive GPU kernels (based on a prior performance model [5]). IPC in the profiling phase can be used to tell the two types of kernels apart. The compiler uses some default values for unknown (i.e., "?") fields; currently, three fields could be unknown: banks, dimension, and concurrency factor.

MSL simplifies porting of GPU programs. For a new GPU, given the MSL specification for its memory system, the PORPLE placer could help determine the appropriate data placements accordingly.

Architects or users could find out the parameters of memory from manuals, or through detection microkernels. PORPLE has a collection of microkernels which users can use to detect some latency parameters, which are similar to those used in prior studies [6], [7]. Users can add more of such microkernels into the library of PORPLE. PORPLE is equipped with a Graphic User Interface (GUI) for users to enter the memory parameters and organizations with ease, from which, MSL specifications are automatically generated.

3 Placer: Performance Modeling and Placement Search

The Placer in PORPLE has three components: one for assessment of the quality of a data placement plan for a kernel, one for search for the best placement plan, and one for online profiling. (Here, a placement plan indicates on which software manageable memory each of the arrays in a kernel is put.) We next explain each of the three components.

3.1 Lightweight Performance Modeling

The first component of the Placer is a performance model, through which, for a given data placement plan and data access patterns (or traces), the Placer can estimate the memory throughput, and hence assess the quality of the plan.

To that end, the Placer tries to determine the number of transactions needed by all the accesses to each array under a given data placement plan. It is simple if there is no memory hierarchy: Based on the data access patterns and the serialization conditions, the Placer can directly compute the number of required transactions. But when there is a memory hierarchy, the Placer has to determine at which level of memory a request can be satisfied. We use the model of reuse distance to address the problem, thanks to its special appeal for quick estimation of cache performance—PORPLE has to conduct many such estimations at runtime to search for the appropriate data placement.

3.1.1 Reuse Distance Models

Reuse distance is a classical way to characterize data locality [8]. The reuse distance of an access A is defined as the number of distinct data items accessed between A and a prior access to the same data item as accessed by A. For example, the reuse distance of the second access to "b" in a trace "b a c c b" is two because two distinct data elements "a" and "c" are accessed between the two accesses to "b". If the reuse distance is no smaller than the cache size, enough data have been brought into cache such that A is a cache miss. Prior studies have shown that that relation is effective for estimating miss rates even for set-associative caches [9], [10].

What PORPLE builds, from the data access patterns of an array, is a reuse distance histogram, which records the percentage of data accesses whose reuse distances fall into each of a series of distance ranges. Cache miss rates can then be estimated as the sum of the heights of all the bars appearing on the right-hand side of the cache size in the histogram. When multiple arrays share a single cache, we consider that each of the arrays gets an equal portion of the cache. The approach is not very precise, but is simple and efficient and has been shown to work reasonably well in prior practices [11]. In one run of a GPU program, PORPLE only needs to construct the histogram for a GPU kernel once, which can be used for many times in that run for estimating cache performance of all possible data placements during the search by PORPLE. With cache miss rates estimated, PORPLE can then tell the portions of accesses to an array that get a hit at each level of a memory hierarchy.

Our construction of reuse distance histograms follows the prior mature techniques, from affine reference patterns [12], and reference traces [13] for irregular accesses. Construction from a trace has a near-linear time complexity [13]; construction from a pattern is even faster. Overall, the time overhead is only a small portion of the online profiling process. The collection of the trace could take some time, which will be discussed in the online profiling part in Section 3.3.

3.1.2 Assessment of Placement Quality

After figuring out what and how many accesses happen on each type of memory, PORPLE converts the numbers into the numbers of transactions by examining the access patterns with the serialization conditions of the memory. Let N_{ij} be the number of memory transactions of array i that happen on memory whose ID equals j. Based on N_{ij}, PORPLE can assess the quality of the data placement plan through a performance model.

Unlike prior heavyweight models [5], [14], [15] that focus on accuracy, PORPLE introduces a light performance model easy to build and quick to use, important for online usage. The model is path-based. It estimates the amount of time taken by data transfers over each of the data transfer paths in the system. On NVIDIA Kepler GPUs, for instance, there are three data transfer paths: one between a core and the global memory, one between a core and the texture memory, and one between a core and the constant memory. Note that the first also covers accesses between a core and the shared memory since those transfers take a part of that first path. Similarly, the second also covers accesses to and from the read-only cache. Because the three paths transfer data concurrently, the performance model uses the maximum of the times estimated on those paths to assess the quality of a data placement.

Specifically, let P be the set of paths, $A(p)$ be the set of arrays whose accesses take path p, N_{ij} be the number of memory transactions of array i that happen on memory whose ID equals j, PORPLE assesses the quality (denoted as T) of a data placement plan through the following performance model:

$$T = \max_{p \in P} T(p);$$
$$T(p) = \sum\nolimits_{i \in A(p)} \sum\nolimits_{j \in memHier(i)} N_{ij} * T_j * \alpha_j.$$

The second formula computes the total time taken by data accesses through one of the paths. The inner summation estimates the total time that accesses to array i incur and the outer summation sums across all arrays going through path p. In the formula, $memHier(i)$ is the memory hierarchy that accesses to array i go through and T_j is the latency of a memory transaction on memory j. The parameter α_j is the concurrency factor of the memory, which takes into account that multiple memory transactions may be served concurrently. For instance, multiple memory transactions can be served concurrently on the global and texture memory (hence for them, $0 < \alpha_j < 1$), while only one memory transaction on the constant memory ($\alpha_j = 1$). The values of concurrency factor are given in the MSL specification; in our experiments, we use 1 for constant memory and 0.2 for other types of memory. That comes from some empirical studies. On *particlefilter* and *bfs*kernels running on Tesla K20c, for instance, when the factor value is fixed to 1 for the constant memory, PORPLE selects different data placements as the concurrency factor value changes from 0 to 1 for the other types of memory. The speedups are 1.18X and 1.25X respectively when the factor value is between 0 and 0.12, 2.11X and 1.25X when the value is between 0.12 and 0.15, and 2.11X and 1.32X when the value is greater than 0.15. Our experiments show that 0.2 gives the overall best results.

3.1.3 Discussion

We acknowledge that the memory performance model could be more sophisticatedly designed. One factor that is not fully considered is the overlapping between different memory accesses and between a memory access and computation. Such overlapping is especially common for GPU thanks to its massive parallelism. However, we note that the use of concurrency factor in the formula offers a simple remedy to the limitation of our model. For instance, a smaller value of the concurrency factor for memory-intensive programs reflects the fact that more memory transactions are likely to overlap in such program executions.

Although the remedy is rough, it suits the purpose of this work by keeping the model simple and fast to use; more sophisticated designs would easily add much more complexity and overhead, hurting the runtime efficiency of PORPLE and its practical applicability. In our experiments, we find that the simple model works well in telling the relative quality among different data placement plans. The intuition is that even though the model may not be accurate, it is enough for ranking the quality of different data placements in most of the time. Moreover, although the formula uses latency but not memory bandwidth, GPU latency often correlates with bandwidth: A memory with a low latency often has a high bandwidth. In cases where latency ratio does not correlate with bandwidth ratios, one could use different concurrency factors (determined empirically) to capture the different influence from the different bandwidths.

3.2 Search for Appropriate Placements

With the capability to determine the quality of an arbitrary data placement plan, the Placer may find the best placement by enumerating all possible plans. However, due to interferences of different arrays in cache, the inter-dependence of arrays performance causes a combinatorially large plan space. Better search algorithms are needed to do the search efficiently.

In the current PORPLE, we implement a hybrid search scheme. It consists of two search algorithms: the branch-and-bound algorithm, and a greedy algorithm. The branch-and-bound algorithm does a parallel depth-first search over a tree. Each tree node represents one way to place an array, and a path from the root to a leaf of the tree represents one placement plan. By maintaining the minimum latency of all visited plans, the algorithm may save some search time by avoiding (part of) some unpromising paths. This algorithm guarantees to find the placement that gives the best result upon the performance model. However, its worst time complexity is exponential to the number of arrays.

The greedy algorithm we propose offers a solution more scalable than the branch-and-bound algorithm. The greedy algorithm takes two heuristics to prioritize the types of memory and data arrays in the placement process. The first heuristic is related with the properties of constant memory over texture memory. There are two scenarios in which constant memory is a better choice than texture memory. The first is when all threads in a warp access the same memory location; the value of the location would be broadcast to all those threads. The second is when too much contention exists in the texture cache. By putting some arrays into the constant memory, the contention gets reduced. Therefore, one may give constant memory a higher priority than other types of memory in the placement process. The second heuristic is that different arrays differ in their optimization potential, and hence one may prioritize the arrays in the placement as well.

Procedure 1 outlines the greedy algorithm. Lines 2 and 3 place the appropriate arrays into constant memory. Lines 5 to 8 estimate the potential benefit of optimizing the placement of each of the remaining arrays (ignoring shared cache contention). Lines 9 to 12 then place those arrays one after one in an descending order of their potential.

Procedure 1. Greedy Algorithm for Searching for Appropriate Data Placements

1: *//$\delta_x(A_i)$ is the extra memory performance of array A_i (i.e., reduction of memory transactions) when it is placed on memory x ("c","t","s","r") versus on global memory.*
2: *currState* = NULL
3: Calculate $\delta_c(A_i)$ for each array (c for constant memory).
4: Sort arrays in descending order of $\delta_c(A_i)$ (the larger the better), and place them into the constant memory until full.
5: Update *currState by indicating the placements*
6: **for** each of the remaining arrays A_i **do**
7: Calculate $\delta_t(A_i)$, $\delta_s(A_i)$, $\delta_r(A_i)$ (t for texture mem; s for shared mem; r for loading through read-only cache).
8: *potential*$(A_i) = \max(\delta_t(A_i), \delta_s(A_i), \delta_r(A_i))$
9: **end for**
10: **for** each of the remaining arrays in a descending order of *potential* **do**
11: *Find the best placement of the array under currState;*
12: *Update currState by marking the placement*
13: **end for**

3.3 Online Profiling

PORPLE does a lightweight on-line profiling, for two purposes. The first is to find out array sizes, which is useful for

the Placer to determine the best data placements. The second is to complement the capability of the compiler in data reference analysis. When PORPLE-C cannot find out the data access patterns (e.g., on irregular programs), it tries to derive a CPU profiling function that keeps the kernel's data access patterns. In cases when the automatic derivation fails, it asks programmers to provide such a function. The function comes with some recording instructions. When the function is invoked at run time, these instructions generate a data access trace, including whether a memory access is a write or read, the array ID and element index, and the GPU thread ID associated with that access.

The profiling happens on the host thread before the GPU kernel gets invoked. Its overhead must be strictly controlled. PORPLE uses two techniques to do so.

First, if the kernel contains a loop, the CPU function only executes the first ten iterations. It is based on the assumption that the truncated iteration space of the first thread block is enough to provide a reasonable approximation of the memory access pattern of the whole workload, which is confirmed by the evaluation on a diverse set of benchmarks in Section 5. We could additionally discard from the function the computations that are irrelevant to data accesses, which is not yet implemented in PORPLE-C.

Second, we employ active learning to further reduce the number of instructions to profile. The CPU function derived from a GPU kernel adds an outmost loop to the kernel body, with each iteration corresponding to the execution of one GPU thread. Using active learning allows PORPLE to profile only a small number of iterations of the loop while still capturing the important access patterns.

Active learning is a technique from machine learning, used for collecting new data points based on the current observations. We use the technique to decide whether we need to profile more iterations of the loop. As the online profiling is mainly to derive the data access patterns—more specifically, reuse distance histograms—of the kernel, the desired design of using active learning is to stop profiling when profiling more does not change the reuse distance histogram much. However, that would require many rounds of calculations of reuse distance histograms, causing large overhead. Through experiments, we find that the average of accessed array indices could serve as a good low-cost clue. Our observation is that if a new iteration causes little change to the average of accessed array indices, adding the data accesses of that iteration usually has little influence on the reuse distance histogram.

The heuristic can be more clearly understood through the pseudo code in Procedure 2. We first explain the cases when there are no data dependences across threads. The code in Procedure 2 outlines the basic algorithm for the active learning-based profiling. The two loops (Lines 2 and 4) are the loops added that surround the kernel body; the outer loop corresponds to all warps, while the inner loop corresponds to the threads within a warp. Lines 6, 7, and 10 represent the code derived directly from the original GPU kernel. Line 7 shows an example access to an element in array y. Line 8 is the profiling instruction, which records the info of that access into a trace buffer for the later analysis.

The other lines in Procedure 2 are related with the active learning. Lines 3, 5, 9, and 11-16 show the code for active learning across the within-a-warp iterations. It uses *sumIndex* to record the sum of the indices of elements accessed in the current iteration. It then checks whether this iteration causes large changes to the average of the accessed indices. If not, it moves to profile the next warp's execution. In the same way, the active learning works across warps as Lines 1 and 18-23 show.

Although the simple heuristic may not faithful reflect changes in reuse distance histograms for some special cases, it turns out to work well in all our tested cases (Section 5). The code in Procedure 2 is for cases when there are no inter-thread data dependences. When there are, the pruning of iterations for profiling could be problematic. The dependences typically could exist at two levels: among threads in a warp, and across an entire thread block. The active learning-based pruning is disabled at a level if there are data dependences at that level. In the worst case, PORPLE profiles the executions of an entire thread block.

Procedure 2. Profiling based on Active Learning

```
1:  prevAvg1_g = 0;prevAvg2_g = −1;
2:  for i ← 0 to NumOfWarps do
3:    prevAvg1_l = 0; prevAvg2_l = −1;
4:    for j ← 0 to WarpSize do
5:      sumIndex = 0;
6:      ...                                  ▷ ommitted kernel code
7:      ... = y[index] + ...;                ▷ a kernel statement
8:      log info of y[index] into a trace buffer;
9:      sumIndex+=index;      ▷ added at each array access
10:     ...                                  ▷ ommitted kernel code
11:     curAvg_l=(prevAvg1_l*j+sumIndex)/(j+1);
12:     if IsStable(curAvg_l, prevAvg1_l, prevAvg2_l) then
13:       break;
14:     else
15:       prevAvg2_l=prevAvg1_l; prevAvg1_l=curAvg_l;
16:     end if
17:   end for
18:   curAvg_g = (prevAvg1_g*i+curAvg_l)/(i+1);
19:   if IsStable(curAvg_g, prevAvg1_g, prevAvg2_g) then
20:     break;
21:     else
22:       prevAvg2_g=prevAvg1_g; prevAvg1_g=curAvg_g;
23:   end if
24: end for
25: function IsStable(cur, pre1, pre2)
26:   δCur = |cur - pre1|;
27:   δPre = |pre1 - pre2|;
28:    if (δCur < 0.005*pre1 || 0.995*δPre < δCur< 1.005*δPre)
       then
29:     return true;
30:   else
31:     return false;
32:   end if
33: end function
```

The online profiling is used only when the kernel is invoked repeatedly for many iterations, which is typical for many real-world irregular applications we have examined. For instance, an N-body simulation program simulates the position change of molecules through a period of time; the kernel is invoked periodically at specific number of time steps. The one-time profiling overhead can be hence outweighed by

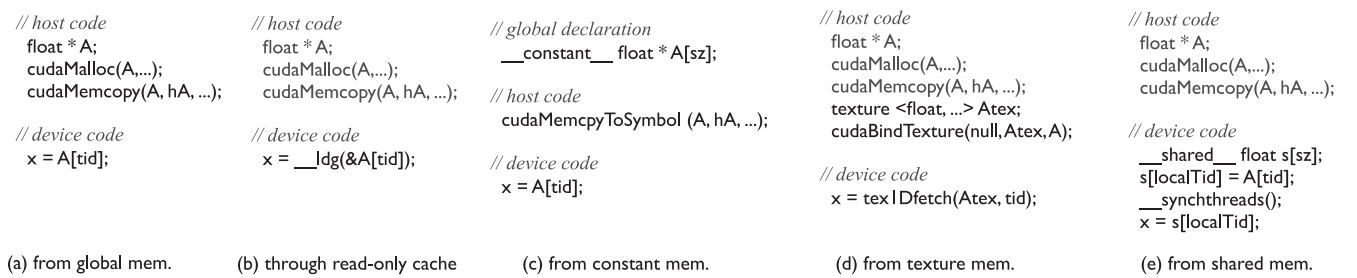


Fig. 5. Codelets in CUDA for accessing an element in an array A.

the benefits from the many invocations of the optimized kernel.

Profiling on GPU rather than CPU is a posisble alternative. It can better capture the warp scheduling effects. One complexity is that for irregular kernels (e.g., those containing indirect memory accesses like A[D[tid]]), the data (e.g., the array D in our example) would need to be copied to GPU for the profiling to take effect. Otherwise, the data access patterns cannot be observed. Moreover, there is kernel launching overhead. We use CPU-based profiling for it can avoid such complexities. A carefully designed hybrid scheme could possibly get the best of both worlds, which is left for future studies.

Active learning has been used for mapping parallel programs to heterogeneous computing devices [16] and performance modeling [17]. We are not aware of a prior use of it for online program profiling.

4 Code Staging through PORPLE-C

When an array is put into different types of memory, the syntax needed to access the array is different. For instance, Fig. 5 shows the syntax for accessing an element in array A in four scenarios. As shown in Fig. 5a, using a simple indexing operator "[]" is enough to access an element of A if it resides on global memory. But to make sure that the access goes through the read-only cache, one needs to instead use the intrinsic "__ldg()", shown in Fig. 5b. The code is more substantially different when it comes to texture memory and shared memory. For texture memory, as Fig. 5d shows, besides some mandatory intrinsics (e.g., "tex1Dfetch"), the access has to go through a texture reference defined and bound to A in the host code. For shared memory, because the allocation of shared memory has to be inside a GPU kernel, the kernel must have code that first declares a buffer on shared memory, and then loads elements of A into the buffer; the accesses to elements in A also need to be changed to accesses to that buffer.

For a program to be amenable to the runtime data placement, it must be *placement-agnostic*, meaning at runtime, the program is able to place data according to the suggestions by PORPLE, and at the same time, run correctly regardless which part of the memory system the data end up on. Runtime code modification through just-in-time compilation or binary translation could be an option, but complex.

Our solution is PORPLE-C, a compiler that generates placement-agnostic GPU programs through source-to-source translations. The solution is a combination of coarse-grained and fine-grained versioning. The coarse-grained versioning creates multiple versions of the GPU kernel, with each corresponding to one possible placement of the arrays. The appropriate version is invoked through a runtime selection based on the result from the Placer.

When there are too many possible placements, the coarse-grained versions are created for only some of the placements (e.g., five most likely ones); for all other placements, a special copy of the kernel is invoked. This copy is fine-grained versioned, which is illustrated by Fig. 6d. The figure shows the code generated by the compiler from a statement "A1[j]=A0[i]". Because the compiler is uncertain about where $A0$ will be put at runtime, it generates a switch statement to cover all possible cases. The value checked by the statement, "memSpace[A0_id]", is the placement of that array determined by the execution of the Placer. The determination mechanism is implemented by the function "PORPLE_place" shown in Fig 6b. The compiler assigns a unique integer as the ID number of each array in the program (e.g., $A0\_id$ is the ID number for the array $A0$). Each case statement corresponds to one possible placement of the array; the compiler produces the suitable statement to read the element for each case. A similar treatment is given to the access to array $A1$, except that the compiler recognizes that there are only two data placement options for $A1$, either in global or shared memory—the alternatives cannot happen because of write limitation.

We now further describe each of the five case statements for the access to $A0$ shown in Fig. 6d. Through this example, we explain how the compiler makes a GPU program placement-agnostic in general.

1) Global Memory. The first two case statements correspond to the global memory without or with read-only cache used. They are straightforward.

2) Texture Memory. The third is when $A0$ is put onto texture memory. In that case, accesses have to go through a texture reference rather than the original array. The compiler hence generates a global declaration of a texture reference $A0tex$. Fig. 6a shows such a declaration and also the declarations for other arrays in the program. The compiler automatically avoids generating such declarations for arrays (e.g., $A1$ in our example) that it regards impossible to be put onto the texture memory. The binding of a texture reference and its corresponding array is done when the "PORPLE_place" function decides to put the array onto texture memory, as shown in Fig. 6b.

3) Shared Memory. The fourth case statement is when $A0$ is put onto shared memory. Two complexities must be addressed in this case: The data have to be copied from global memory into the shared memory and sometimes also copied back; the index of an element in shared memory differs from its index in global memory.

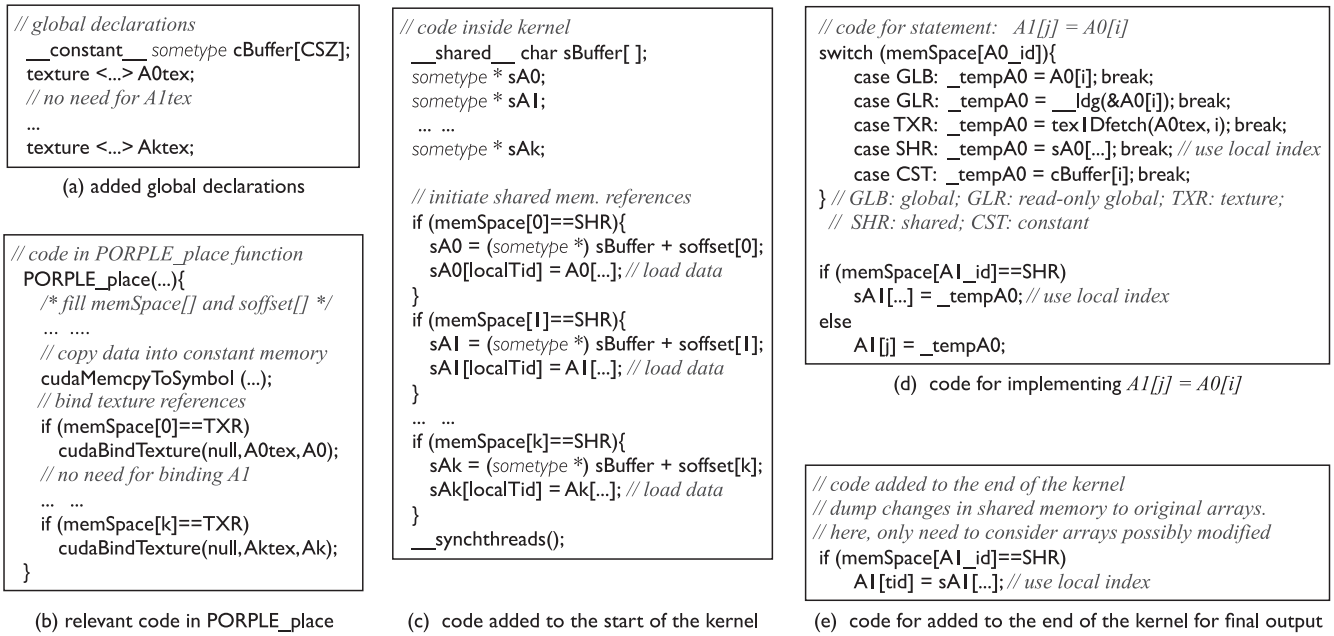


Fig. 6. Generated placement-agnostic code.

Fig. 6c shows the code the compiler inserts to the beginning part of a GPU kernel function to support the case of shared memory. It starts with the declaration of an array allocated onto shared memory. That array will be used as the buffer to store the elements of arrays suitable for using shared memory. The size of the array is determined by one of the arguments in the kernel call. Before the kernel call, the argument is assigned a value computed by the Placer at runtime. The computation is based on the data placements Placer finds. Meanwhile, the Placer tries to ensure that the size does not lower the number of concurrently runnable thread blocks (called GPU *occupancy*) compared to the original GPU program.

It is allowed for *sBuffer* to contain the elements of multiple arrays. To save the address lookup time, the compiler inserts the declaration of a pointer (e.g., *sA*0 for *A*0) for each array that is possible to be put into shared memory. The pointer is then set to the starting position of the corresponding array in *sBuffer*. By this means, the kernel can access the elements through that pointer.

The code in the "if" statements in Fig. 6c also loads array elements from global memory into shared memory. Based on the affine expressions of array accesses in the kernel, the compiler builds up a one-to-one mapping between the indices of the elements in the original array and their indices in shared memory. It uses such a mapping to load the elements of the array into the corresponding location in shared memory. This mapping is also used when the compiler generates the statements in the kernel function to access the correct elements in the shared memory.

At the end of the kernel, the compiler inserts some code to copy data from shared memory back to the original array on global memory, if the data could be modified in the kernel, as illustrated by Fig. 6e.

4) Constant Memory. The final case is when the array is put into constant memory. The treatment is similar to shared memory. Fig. 6c illustrates the code for putting an array into the constant memory through the call of "cudaMemcpyToSymbol" function. Multiple arrays could share the constant memory in a way similar to that of shared memory; details omitted.

5) Compiler Implementation. The implementation of the compiler is based on Cetus [18], a source-to-source compiler. The input is CUDA code. As a prototype, the compiler cannot yet handle all kinds of CUDA code complexities; but with some minor manual help, it is sufficient for proving the concept. If the input program already has some arrays put onto memory other than global memory, PORPLE by default respects the optimizations performed by the programmer and keep them unchanged; it optimizes the placement of the data arrays only if they are on global memory. The compiler follows the following steps to generate the placement-agnostic form of the code.

Step 1: find all arrays that are on global memory in the kernel functions, assign ID numbers, and generate access expressions for the arrays;
Step 2: identify the feasible placement options for each array to avoid generating useless code in the follow-up steps;
Step 3: create global declarations for the constant buffer and texture references (as illus. by Fig. 6a);
Step 4: customize PORPLE_place function accordingly (as illus. by Fig. 6b);
Step 5: insert code at the start and end of each kernel function and change data access statements (as illus. by Figs. 6c, 6d, 6e).

To make it simple to generate code for accessing a new type of memory, PORPLE defines five ways of memory accesses: through direct indexing (global memory-like), through binding on host (texture memory-like), through host declared buffer (constant memory-like), through kernel declared buffer (shared memory-like), and through special intrinsics (read-only global memory-like). There are some fields that users can fill for each of the five ways, including the keywords to use to make the declaration, the intrinsics to use for access, and so on, based on which, the PORPLE-C

TABLE 1
Benchmark Description

Benchmark Kernels	Source	Description	Irregular
mm	SDK	matrix multiplication	N
trans	SDK	matrix transpose	N
convolution	SDK	signal filter	N
kmeans	Rodinia	kmeans clustering	N
particlefilter	Rodinia	particlefilter	Y
cfd	Rodinia	computational fluid	Y
reduction	SHOC	reduction	N
fft	SHOC	fast Fourier transform	N
scan	SHOC	scan	N
sort	SHOC	radix sort	N
triad	SHOC	stream triad	N
md	SHOC	molecular dynamics	Y
spmv	SHOC	sparse matrix vector multi.	Y
bfs	SHOC	breadth-first search	Y
glassForce	Lulesh	force computation	Y
glassControl	Lulesh	glass control for element	Y

will try to generate needed code for a kernel to utilize a new type of memory. For memory unlike any of the five, PORPLE provides recommended placement to the programmer, who can then refactor the code accordingly.

5 Evaluation

We conduct experiments on three GPU models to evaluate the effectiveness and portability of the technique.

5.1 Methodology

We evaluate PORPLE on a diverse set of benchmarks shown in Table 1. These benchmarks include all of the level-1 benchmarks from the SHOC benchmark suite [19] that come from various application domains. To further evaluate PORPLE with complicated memory access patterns, we add three benchmarks from the RODINIA benchmark suite [20] and three from CUDA SDK. To test the scalability of PORPLE, we additionally include the two most time-consuming kernels from *lulesh*, a large application from Lawrence Livermore National Lab for modeling hydrodynamics [21]. They use much more (16) arrays than the other benchmarks do.

As shown by the rightmost column of Table 1, seven of the benchmarks have irregular memory accesses. Their memory access patterns highly depend on inputs and can only be known during run-time. Hence, static analysis cannot work for them and online profiling must be employed. They all have a loop surrounding the GPU kernel call. We focus on the optimization of the most time-consuming kernel in each of the benchmarks. To optimize data placement for multiple kernels, PORPLE would need to take into consideration the possibly required data movements across the kernels, which is left for future studies.

TABLE 2
Machine Description

Name	GPU card	OS	Processor	CUDA
C1060	Tesla C1060	Linux-3.11	E5640	5.5
M2075	Tesla M2075	Linux-2.6	Intel Xeon(R) X5672	5.5
K20c	Tesla K20c	Linux-3.13	Intel Xeon(R) E5-1607v2	6.5

We evaluate PORPLE on three different machines with diverse GPU hardware and runtime environment shown in Table 2. The GPU cards have quite different memory hierarchies. Most notably, C1060 does not have any data cache for global memory accesses. M2075 has a two-level data cache for global memory. K20c has a L2 cache for global memory. Each SM on it has a read-only L1 data cache. However, it is used for global memory accesses only if the accesses are to read-only data through some intrinsics. One transaction for a global load is 128 bytes on M2075, but 32 bytes on K20c.

Since the specifics of some of the memories are undisclosed, we use a tool published by Wong [7] to obtain the memory specification for each machine. The tool runs a set of microkernels on each machine, and measures cache size and latency for each type of memory. The memory latency results are summarized in Table 3.

We compare PORPLE with the state-of-the-art memory selection algorithm published previously [1]. In that work, data placement decisions are made with several rules. These rules are based on read/write patterns, loop-based temporal locality, and status of memory coalescing that are determined through some static analysis of the kernel code. For data arrays whose access patterns cannot be inferred through the static analysis, this algorithm simply leaves them in global memory space. We call this algorithm the *rule-based approach*. In addition, we find the optimal data placement through offline exhaustive search, which produces the best speedup a data placement method can achieve. The search time varies across the kernels. On glassForce with a single small input on one architecture, for example, the time is over two hours.

We repeat each time measurement experiment for 10 times. We report the average performance along with the error bars in the overall speedup results. All the reported speedups in this section are based on the formula *originalTime*/*newTime*, where *originalTime* refers to the total execution time taken by all invocations of the original kernel, and *newTime* includes the time taken by all invocations of the optimized kernel plus all optimization overhead (profiling time, time taken by Placer, etc.).

TABLE 3
Memory Latency Description

Machine Name	Constant	cL2	cL1	Global	gL2	gL1	Read-only	Texture	tL2	tL1	Shared
Tesla K20c	250	120	48	345	222	N/A	141	351	222	103	48
Tesla M2075	360	140	48	600	390	80	N/A	617	390	208	48
Tesla C1060	545	130	56	548	N/A	N/A	N/A	546	366	251	38

cL1 and cL2 are L1 and L2 caches for constant memory. gL1 and gL2 are L1 and L2 caches for global memory. tL1 and tL2 are L1 and L2 caches for texture memory.

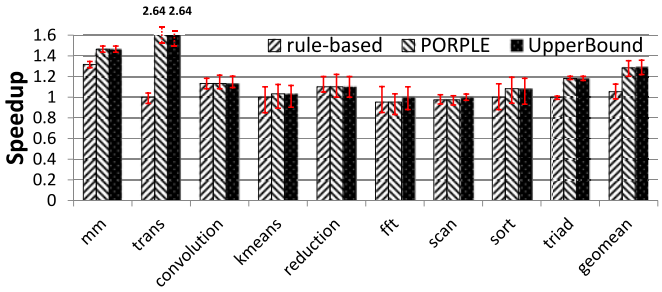


Fig. 7. Speedup (avarage with errors bar) of regular benchmarks on Tesla K20c.

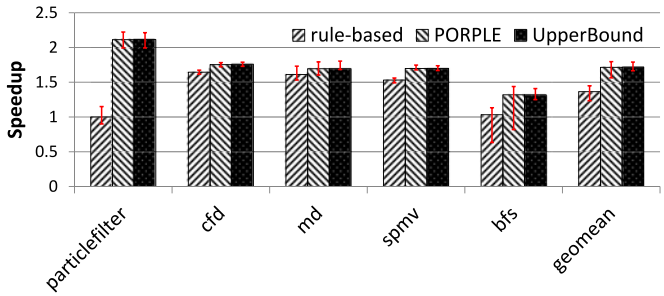


Fig. 8. Speedup (average with error bars) of irregular benchmarks on Tesla K20c.

5.2 Results of Regular Benchmarks

Fig. 7 shows the performance results for regular benchmarks on Tesla K20c. PORPLE provides on average 22 percent speedup and obtains almost all the potential (23 percent) by finding out the best data placement. The rule-based approach only provides 5 percent performance improvement. We observe that on *kmeans* and *scan*, the data placement strategies in the original programs are either close to optimal, showing that the programmers have already taken advantage of their easily analysable memory access patterns and made good placement decisions. Hence, there is little potential for further improvement. On them, PORPLE and the rule-based approach both find the optimal placement strategies, which perform similarly as the original programs do.

Benchmarks *mm*, *trans* and *triad* show much larger speedups (1.18X to 2.64X). For instance, PORPLE outperforms the rule-based approach with 45 and 18 percent additional speedups on *trans* and *triad*. Our investigation reveals that the rule-based approach favors global memory because of its limited capabilities to characterize memory access patterns and map them to diverse memory systems. In particular, for arrays with coalesced accesses and little temporal reuse, the rule-based approach always places them into global memory. In contrast, PORPLE's performance model captures the fact that texture cache can be used instead of L1/L2 cache for those arrays, even if those arrays are linearly accessed. Moreover, the path-based performance model helps PORPLE select the placements that can balance the different data transfer paths. Program *fft* shows a little slowdown over the original program. The inaccuracy in the cache performance estimation causes the runtime placer to select a suboptimal placement.

The runtime overhead for PORPLE is marginal (around 1 percent) on regular benchmarks, and hence they are not reported. As described in Section 4, for these codes that are statically analysable, PORPLE performs offline transformation to enforce the placement strategy, which significantly reduces runtime overhead. For those benchmarks whose data placement strategies cannot be fully determined using the offline approach, PORPLE can still use static analysis to exclude some data placement options and reduce search space. This results in great reduction of runtime overhead.

The results on the other two machines are similar to those on K20c. We ommit them for the interest of space.

5.3 Results with Irregular Benchmarks

5.3.1 Speedup

Fig. 8 shows the results on the irregular benchmarks on K20c. The complex memory access patterns of these benchmarks make them difficult to optimize by both static analysis and manual efforts. The original programs choose data placement strategies substantially inferior to the optimal. PORPLE again gives near optimal performance, on average 1.68X and up to 2.17X (for *particlefilter*) speedups over the original ones.

We observe that the rule-based approach works well for some benchmarks, but fails to find the optimal dataplacement. For example, it identifies optimal placement strategy for some important arrays in *spmv*, *cfd* and *md*, but fails for other arrays. For *particlefilter* and *bfs*, the rule-based approach shows much smaller speedups compared to the optimal one, because it inappropriately uses texture memory and global memory for some arrays. For *particlefilter*, for instance, there is an array named CDF that can benefit more from constant memory than from texture memory, because its access patterns meet the broadcasting requirements and constant cache can hence be accessed more efficiently than texture cache. However, the rule-based approach tends to use texture memory to favor specific memory access patterns, ignoring potentially benefits of cache hierarchy in constant memory; as many arrays are put onto texture memory, texture cache interferences are severe.

Benchmark *bfs* conducts breadth-first search. It is special in that it allows race conditions to happen. Specifically, all GPU threads read and write array *levels*; even though two threads could access the same data element in that array, *bfs* uses no synchronizations for high efficiency. Such race conditions however do not affect the results because if multiple threads try to assign values to a single data element in *levels*, those values must be identical. After noticing such a property, we add into PORPLE the check for the profit of duplicating such an array under such race conditions. Specifically, it checks the benefit of having the following version: A duplicate of array *levels*, named *levels1*, is created before each kernel call (*BFS_kernel_warp*). In the kernel, all reads to *levels* are changed to reads to *levels1*, while writes remain unchanged. PORPLE then tries to figure out the best placement of arrays of this new version and compares its memory throughput with the throughput of the best placement of the original version. If it is beneficial, it offers the suggestion to the programmer; after confirming the safety of such a transformation, the programmer may refactor the code accordingly. For *bfs*, the transformation is safe, and after the transformation is done, PORPLE puts array *levels*1 into the constant memory, achieving 30 percent extra speedups over the rule-based approach which uses only global memory for the arrays. For the rule-based approach, after

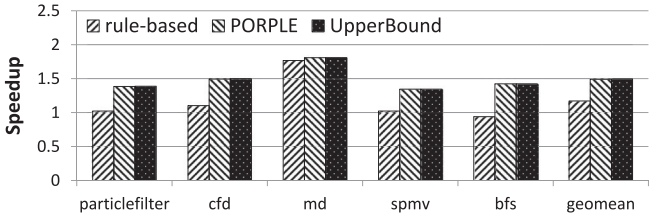


Fig. 9. Speedup of irregular benchmarks on Tesla M2075.

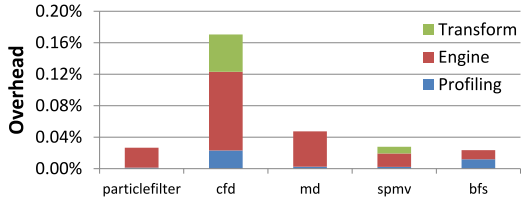


Fig. 11. Overhead breakdown on irregular benchmarks on Tesla K20c.

the same duplication, it achieves 1.18X speedups. It still misses the good placement because although it can put *levels1* on constant memory, it puts all other read-only arrays on texture memory and causing a lot of cache interferences.

Figs. 9 and 10 display the results on M2075 and C1060. PORPLE shows little performance gap from the optimal, and performs much better than the rule-based approach (33 and 60 percent more on M2075 and C1060 respectively). There are two main reasons that could cause disparity of PORPLE from the optimal. First, PORPLE only profiles the memory accesses of part of the threads in order to minimize runtime overhead, which could cause some inaccuracy in the profiling results. Second, PORPLE employs a conservative approach to model cache interference. In some cases, data arrays, when placed into the same memory system, may not cause severe cache interference. However PORPLE may choose to spread them into multiple memory systems based on performance prediction. This could cause non-optimal data placement. The results show that the influence of the factors on the final data placement is marginal.

There are some differences in the performance gains among the different GPUs, due to the differences in their memory latencies and cache hierarchy. For example, L1 cache exists on K20c and M2075, but not on C1060. The global memory access latency on C1060 is also much longer than on the other two GPUs. As a result, we see that some programs (e.g., *particlefilter*, *spmv*, *md*) enjoy larger speedups on C1060 than on other GPUs when PORPLE puts some arrays onto the texture memory. L1 cache exists on K20c but loads from the global memory on K20c do not go through L1 cache. That is why quite large speedups are also seen on K20c. The speedups on M2075 is relatively smaller for its L1 cache helps hide the latency of global memory accesses.

5.3.2 Overhead Breakdown

Fig. 11 reports runtime overhead of PORPLE on Tesla K20c. On average, PORPLE introduces 0.06 percent overhead, which is outweighed by the performance benefits as the speedup results show. The overhead can be decomposed into three parts: profiling, transform and the placement search (denoted as *engine* in the Fig. 11). The transform overhead is due to the runtime checks introduced by PORPLE in the transformed kernels. The overhead of placement search comes from the performance modeling and branch-and-bound search. We observe that the overhead of *engine* is the main source of overhead due to analyze trace and placement search, accounting for average 0.04 percent of total execution time. Programs *spmv* and *cfd* are subject to somewhat higher transformation overhead. For *cfd*, the reason is that the number of fine-grained checks is large due to the large number of arrays. For *spmv*, the reason is that it has a *for* loop and the number of fine-grained checks for one array inside the loop is large.

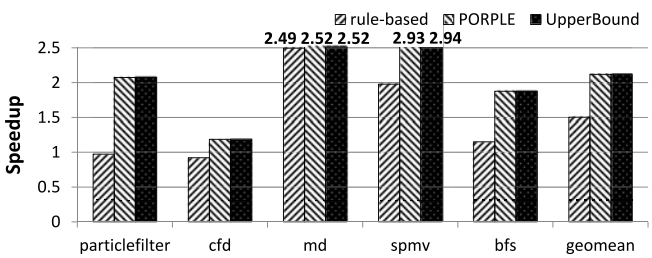


Fig. 10. Speedup of irregular benchmarks on Tesla C1060.

5.3.3 Portability

Table 4 shows the placement decisions made by the rule-based approach and PORPLE. The rule-based approach gives the same data placement on different platforms, because it ignores the many subtle architecture differences across hardware. In contrast, PORPLE explicitly expresses, quantifies, and models diverse memory features across platforms, hence providing much better data placement decisions. For benchmark *spmv*, for instance, on the three machines, PORPLE makes quite different decisions.

To study adaptivity of PORPLE to input changes, we use six different inputs to *particlefilter* and *spmv*, and examine the data placements PORPLE finds and the corresponding performance. For *particlefilter*, we use the input generator in the benchmark to generate inputs with different number of particles. For *spmv*, we use matrix inputs from the University of Florida's sparse matrix database [22].

The results are shown in Figs. 12 and 13. The figures show that in most cases PORPLE outperforms the rule-based approach across the different inputs, with much larger average speedups (72 and 30 percent respectively) over those of the rule-based approach (7 and 24 percent respectively). The only exception is when *spmv* runs on a

TABLE 4
Placement Decisions Made by PORPLE and the Rule-Based Approach

	spmv					particlefilter					
	A0	A1	A2	A3	A4	B0	B1	B2	B3	B4	B5
Rule-Based	T	T	T	T	G	G	S&G	G	G	G	G
PORPLE-C1060	C	T	T	T	G	C	S&G	G	G	G	G
PORPLE-M2075	C	T	G	T	G	C	S&G	G	G	G	G
PORPLE-K20c	C	R	T	R	G	C	S&R	G	T	G	G

T: texture memory, C: constant memory, G: global memory, S: shared memory, R:read-only data cache. Spmv: A0:rowDelimiters, A1:cols, A2:vec, A3:val, A4: out. Particlefilter: B0:CDF, B1:u, B2: arrayX, B3:arrayY, B4:xj,B5:yj.

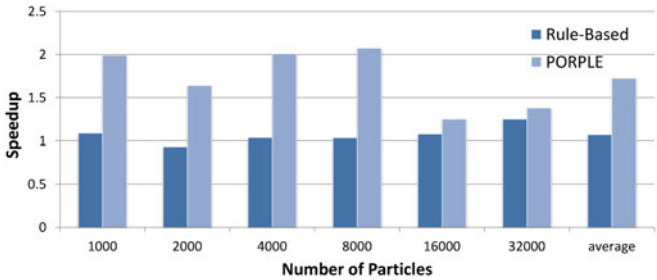


Fig. 12. Speedup across different inputs for particlefilter.

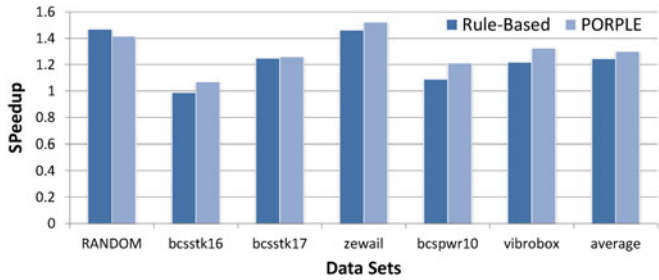


Fig. 13. Speedup across different inputs for spmv.

special input named *random*. For this input, it is difficult to precisely capture the data access patterns, and PORPLE performs slightly worse than the rule-based approach.

5.3.4 Scalability

We conduct a scalability study of PORPLE through a series of experiments on eight kernels that are more sensitive to the runtime overhead of PORPLE than other kernels. The set includes the two kernels from *lulesh* that use much more kernels than other kernels do, four other irregular benchmarks and two regular benchmarks from the set of kernels listed in Table 1.

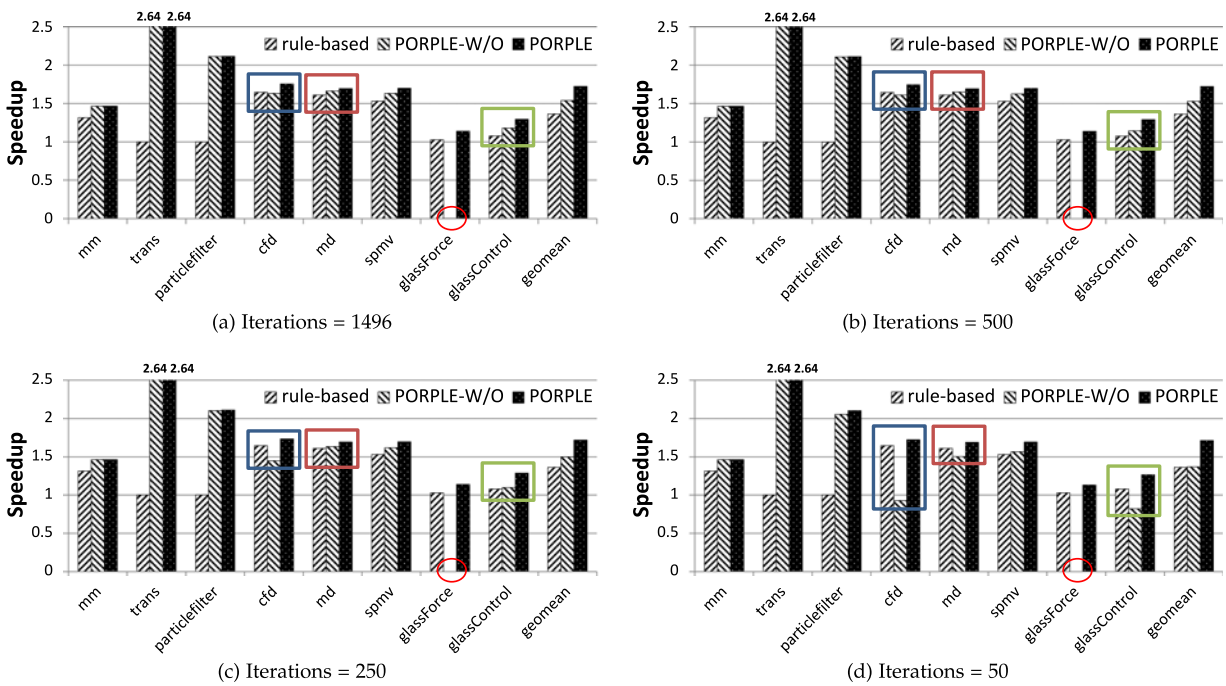
The main overhead of PORPLE resides in the online profiling part, the search for appropriate data placements, and runtime checks in the GPU kernels. Among them, the first two parts happen only once in a program execution regardless how many times the GPU kernel gets invoked, and the third part occur in every GPU invocation. Experiments however show that the first two parts are actually the dominant sources of overhead.

In this scalability study, we change the number of iterations of the loop that surrounds a kernel invocation. As the number of iterations decreases, the influence of the runtime overhead becomes more visible. We compare the performance of PORPLE with the rule-based method, as well as an earlier version of PORPLE [2] (denoted as "PORPLE W/O") that does not use the aforementioned optimizations to profiling and search (active profiling and hybrid scalable search).

Fig. 14 reports the performance comparison on Tesla K20c (results on other GPUs show similar trends). The number of iterations starts with 1,496 (a number used in the original lulesh application) and decreases to 500, 250, and 50. Observations are as follows.

(1) For the leftmost four benchmarks (*mm*, *trans*, *particlefilter*, and *spmv*), the influence of the runtime overhead across the number of iterations stays almost the same for all the three methods. That is because the four benchmarks are either regular or use very few arrays. The profiling and search time weighs little in the overall running time. PORPLE and PORPLE W/O performs similarly well except that on *spmv*, PORPLE gains slightly more benefits thanks to the reduction of profiling overhead by active profiling.

(2) For the other four benchmarks, the influence of the runtime overhead varies significantly for PORPLE W/O when the number of iterations decreases. On *glassForce*, the large search time of the branch-and-bound method in PORPLE W/O makes the kernel



(a) Iterations = 1496

(b) Iterations = 500

(c) Iterations = 250

(d) Iterations = 50

Fig. 14. Speedups on Tesla K20c when the number of kernel invocations changes.

TABLE 5
Overheads of PORPLE-W/O and PORPLE

Methods	PORPLE-W/O (sec)			PORPLE (sec)			Speedup	
Benchmarks	profiling	engine	transform	profiling	engine	transform	profiling speedup	engine speedup
particlefilter	0.000043	0.0024	0	0.000024	0.000450	0	1.79	5.33
spmv	0.000055	0.00412	0	0.000037	0.00028	0	1.48	14.71
cfd	0.000146	0.00867	0.000014	0.000047	0.00075	0.000014	3.11	11.56
md	0.000193	0.05892	0	0.000054	0.00093	0	3.57	63.35
glassForce	0.00006	542.418	0.00033	0.000037	0.00036	0.00033	1.62	1506716.67
glassControl	0.000012	0.00742	0.000066	0.000003	0.000322	0.000066	4	23.04

run much slower than the original version does, even when the kernel is called for 1,496 times. The branch-and-bound search has an exponential complexity. For the 16 arrays in *glassForce*, the search time is unaffordable. PORPLE, through the hybrid search algorithm, overcomes the scalability issue. It gives significant speedups across all the cases even when the number of iterations is reduced to 50.

(3) In all the scenarios, both PORPLE versions give a higher average speedup than the rule-based method, demonstrating the benefits of the better data placements those methods find.

Table 5 reports the breakdown of the runtime overhead on the six irregular programs (the overhead of the two regular ones is less than 1 percent, hence omitted). The active profiling saves profiling time by 1.79X to 4X. On *spmv*, for instance, PORPLE W/O profiles the work of an entire thread block, while PORPLE profiles only the work of two threads per warp (8 warps in total). Similarly, on *md*, PORPLE profiles the work of only some threads (9, 6, 2, 7, 11 threads) in the first 5 (out of 8) warps. The reduced profiling still captures enough information about the data access patterns.

The time savings by the hybrid search algorithm is even greater, ranging from 5.3X to 10^6X. It is because the algorithm reduces the computational complexity from exponential to linear when there are many arrays.

Overall, the two optimizations significantly improve the scalability of PORPLE.

6 Related Work

Data placement on GPU memory has drawn some previous studies, including the rule-based method that the previous section has compared with [1]. In addition, some other studies have investigated data placement on some special type of memory. Ma and others [23] have considered optimal data placements but only for shared memory on GPU. They formulate the problem of placing scalar, data array and sections of data arrays on shared memory as an integer programming problem, and evaluate the idea on one benchmark. PORPLE takes all types of memory into consideration and thoroughly evaluates a diverse set of benchmarks on different machines. Wang and others [24] have designed a heterogeneous main memory consisting of both DRAM and phase change memory for GPU. They propose a compiler-guided initial placement for data arrays and hardware-based fine-grained data migration between the two types of memory. PORPLE offers a portable solution for off-the-shelf GPU cards. Agarwal and others [25] have recently proposed some improved memory page placement policies for future heterogeneous systems with a unified globally-addressable memory.

Some previous work has studied data placement on CPU systems that are equipped with heterogeneous memory architectures. Jevdjic and others [26] have studied the design of CPU with 3D stacked memory. Due to the various technical constraints, especially heat dissipation, the stacked memory has limited size and is managed by hardware like traditional cache. Similarly, some heterogeneous memory designs [27], [28] involving Phase Change Memory have also resorted to hardware-managed data placement.

Several previous efforts have tried to develop some performance models for GPU [5], [6]. For instance, Hong and others [5] have proposed a sophisticated GPU analytical performance model containing a number of parameters. Baghsorkhi and others [14] have proposed a compiler-based performance model, which considers detailed micro-architecture features, such as shared memory bank conflicts and warp divergence. The designs of these models primarily aim at accuracy of the models, suiting offline usage. The performance model in PORPLE is designed to be fast enough for runtime use. It considers cache contention through the reuse distance model.

GPU memory performance has received many attentions [29], [30], [31], [32]. For instance, Yang and others [33] have designed a source-to-source compiler to enhance memory coalescing or shared memory use. Zhang and others [34] have focused on irregular memory references and proposed a pipelined online data reorganization engine to reduce memory access irregularity. Wu and others [35] have provided a formal representation of data reorganization for minimizing non-coalesced memory accesses, and provided the first complexity analysis and propose several efficient reorganization algorithms. Wu and others [36] have recently introduced *SM-centric* transformation to enable software-based spatial management of GPU threads, and showed that it may help improve cache performance of GPU programs. All these studies mainly focus on altering memory access patterns. PORPLE complements them with portable optimizations of data placements.

7 Conclusion

This work gives a comprehensive description of PORPLE, a software framework for portable optimizations of data placement on GPU memory. It consists of a mini specification language, a source-to-source compiler, and a runtime data placer. The language allows an easy description of a memory system; the compiler transforms a GPU program into a form

amenable to runtime profiling and data placement; the placer, based on the memory description and data access patterns, computes on the fly the appropriate placement schemes for the data and places them accordingly. PORPLE is distinctive in being adaptive to program inputs and architecture changes, being transparent to programmers (in most cases), and being extensible to new memory architectures. Experiments show that PORPLE is able to consistently find optimal or near-optimal placements despite the large differences among GPU architectures and program inputs, yielding up to 2.64X (1.72X on average) speedups on a set of regular and irregular GPU benchmarks.

Industry has been continuously inventing new types of memory, exemplified by 3D-stacked memory and Phase Changing Memory. The innovations are expected to lead to even more sophisticated memory systems in future machines. The portable approach proposed in this work has the potential to be extended into a solution for tapping into the power of such future heterogeneous memory systems beyond GPU.

REFERENCES

[1] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 105–118, Jan. 2011.

[2] G. Chen, B. Wu, D. Li, and X. Shen, "Porple: An extensible optimizer for portable data placement on GPU," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2014, pp. 88–100.

[3] G. Chen, B. Wu, D. Li, and X. Shen, "Enabling portable optimizations of data placement on GPU," *IEEE Micro*, vol. 35, no. 4, pp. 16–24, Jul./Aug. 2015.

[4] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters," in *Proc. 10th Int. Symp. Code Generation Optimization*, 2012, pp. 155–164.

[5] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 152–163, 2009.

[6] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *Proc. 2011 IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 382–393.

[7] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Proc. IEEE Symp. Performance Anal. Syst. Softw.*, 2010, pp. 235–246.

[8] A. P. Batson and A. W. Madison, "Measurements of major locality phases in symbolic reference strings," presented at the Conf. Meas. Modeling Comput. Syst., Cambridge, MA, USA, Mar. 1976.

[9] A. J. Smith, "On the effectiveness of set associative page mapping and its applications in main memory management," in *Proc. 2nd Int. Conf. Softw. Eng.*, 1976, pp. 286–292.

[10] Y. Zhong, S. G. Dropsho, and C. Ding, "Miss rate prediction across all program inputs," presented at the 12th Int. Conf. Parallel Archit. Compilation Techn., New Orleans, LA, USA, Sep. 2003.

[11] S. Manegold, P. Boncz, and M. L. Kersten, "Generic database cost models for hierarchical memory systems," in *Proc. 28th Int. Conf. Very Large Data Bases*, 2002, pp. 191–202.

[12] G. C. Cascaval, "Compile-time performance prediction of scientific programs," Ph.D. dissertation, Dept. Comput. Sci., Univ. Illinois Urbana-Champaign, Champaign, IL, USA, 2000.

[13] C. Ding and Y. Zhong, "Predicting whole-program locality with reuse distance analysis," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, Jun. 2003, pp. 245–257.

[14] 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 Practice Parallel Program.*, 2010, pp. 105–114.

[15] J. Sim, A. Dasgupta, H. Kim, and R. W. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *Proc. 17th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 11–22.

[16] W. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather, "Active learning accelerated automatic heuristic construction for parallel program mapping," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation Techn.*, 2014, pp. 481–482.

[17] P. Balaprakash, K. Rupp, A. Mametjanov, R. Gramacy, P. Hovland, and S. Wild, "Empirical performance modeling of GPU kernels using active learning," in *Proc. Symp. Appl. Autotuning HPC*, 2013, pp. 1–14.

[18] S. Lee, T. Johnson, and R. Eigenmann, "Cetus—An extensible compiler infrastructure for source-to-source transformation," in *Proc. 16th Annu. Workshop Languages Compilers Parallel Comput.*, 2003, pp. 539–553.

[19] A. Danalis, et al., "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proc. 3rd Workshop General-Purpose Comput. Graph. Process. Units*, 2010, pp. 63–74.

[20] S. Che, et al., "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.

[21] I. Karlin, et al., "Exploring traditional and emerging parallel programming models using a proxy application," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 919–932.

[22] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[23] W. Ma and G. Agrawal, "An integer programming framework for optimizing shared memory use on GPUs," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 553–554.

[24] B. Wang, et al., "Exploring hybrid memory for GPU energy efficiency through software-hardware co-design," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.*, 2013, pp. 93–102.

[25] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for GPUs within heterogeneous memory systems," in *Proc. 20th Int. Conf. Archit. Support Program. Languages Operating Syst.*, pp. 607–618, 2015.

[26] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? Have it all with footprint cache," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 404–415.

[27] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proc. Int. Conf. Supercomputing*, 2011, pp. 85–95.

[28] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 24–33.

[29] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing memory access patterns for heterogeneous systems," in *Proc. 2011 Int. Conf. High Performance Comput., Netw. Storage Anal.*, 2011, pp. 13:1–13:11.

[30] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs," in *Proc. 22nd Annu. Int. Conf. Supercomputing*, 2008, pp. 225–234.

[31] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 513–522.

[32] 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. Supercomputing*, 2012, pp. 15–24.

[33] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *Proc. 2010 ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2010, pp. 86–97.

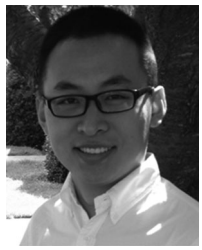
[34] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for GPU computing," in *Proc. 16th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2011, pp. 369–380.

[35] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU," in *Proc. 18th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 57–68.

[36] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations," in *Proc. 29th ACM Int. Conf. Supercomputing*, 2015, pp. 119–130.

Guoyang Chen received the BS degree in computer science from the University of Science and Technology of China, in 2012. He is working toward the PhD degree in the Computer Science Department, NCSU, where he is working with his advisor Prof. Xipeng Shen. He is an IBM Center for Advanced Studies Fellowship Receiver from 2014 to 2016 and a receipt of Cisco Connected-recognition You Accelerate Award in 2015. Prior to joining NC State in 2014, Chen was a PhD student in the Computer Science Department, College of William and Mary. His research interests include compilers, program analysis and optimizations, heterogeneous computing, high performance computing.

Xipeng Shen received the PhD in computer science from the University of Rochester, in 2006. He is an associate professor in the Computer Science Department, NCSU, and IBM Center for advanced studies faculty fellow. His research interests include broad field of compiler and programming systems, with an emphasis on enabling data-intensive high performance computing and intelligent computing through innovations in both compilers and runtime systems. He is a receipt of DOE Early Career Research Award, NSF CAREER Award, and Google Faculty Research Award. Prior to joining NC State in 2014, he was the Adina Allen Term distinguished associate professor in the Computer Science Department, College of William and Mary. He was a visiting researcher at MIT, Microsoft Research, and Intel Labs between 2012 and 2013, and an assistant professor in the College of William and Mary from 2006 to 2012. He is a senior member of the IEEE and an ACM distinguished speaker.



Bo Wu received the BS degree in information and computational sciences and the MS degree in computer science from Central South University in Changsha, China. He received the PhD degree from the College of William and Mary, where he completed a thesis on matching non-uniformity between applications and many-core processors. He is currently an assistant professor of computer science at Colorado School of Mines in Golden. His research focuses on GPU computing, graph analytics, heterogeneous memory systems, and compiler optimization.

Dong Li received the PhD degree in computer science from Virginia Tech, in 2011. He is an assistant professor of electrical engineering and computer science with the University of California, Merced. His research focuses on parallel and distributed systems and applications. The core theme of his research is to study how to enable scalable and efficient execution of enterprise and scientific applications on increasingly complex large-scale parallel systems. His work creates innovation in runtime, architecture, performance modeling, and programming models to solve the challenges on performance, power, and reliability. Throughout his career, he has won several awards, including NSF CAREER award, ORNL/CSMD distinguish contributor award, and STINT scholarship award. He is a member of the IEEE and ACM.