2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

PORPLE: An Extensible Optimizer for Portable Data Placement on GPU

Guoyang Chen
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
Email: gchen11@ncsu.edu

Bo Wu
Department of Electrical Engineering and Computer Science
Colorado School of Mines
Golden, CO, USA
Email: bwu@mines.edu

Dong Li
Oak Ridge National Laboratory
Oak Ridge, TN, USA
Email: lid1@ornl.gov

Xipeng Shen
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
Email: xshen5@ncsu.edu

***Abstract*—GPU is often equipped with complex memory systems, including global memory, texture memory, shared memory, constant memory, and various levels of cache. Where to place the data is important for the performance of a GPU program. However, the decision is difficult for a programmer to make because of architecture complexity and the sensitivity of suitable data placements to input and architecture changes.**

This paper presents PORPLE, a portable data placement engine that enables a new way to solve the data placement problem. PORPLE 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, identifies on the fly appropriate placement schemes for 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. Our experiments on three types of GPU systems show that PORPLE is able to consistently find optimal or near-optimal placement despite the large differences among GPU architectures and program inputs, yielding up to 2.08X (1.59X on average) speedups on a set of regular and irregular GPU benchmarks.

***Keywords*-GPU; cache; compiler; data placement; hardware specification language;**

I. Introduction

Modern Graphic Processing Units (GPU) rely on some complex memory systems to achieve high throughput. On an NVIDIA Kepler device, for instance, there are more than eight types of memory (including caches), with some on-chip, some off-chip, some directly manageable by software, and some not. They each have their own sizes, properties, and access constraints. Studies have shown that finding the suitable kinds of memory to place data—called the *data placement problem*—is essential for GPU program performance [12]. But due to the complexity of memory and its continuous evolvement, it is often difficult for programmers to tell what data placements fit a program. For some programs, the suitable placements differ across the program inputs, making the placement even more difficult to do.

There have been some efforts to addressing the problem. Some of them use offline autotuning, which tries many different placements and measures the performance on some training runs [29]. This approach is time-consuming, and cannot easily adapt to the changes in program inputs or memory systems. Some others use some high-level rules derived empirically from many executions on a GPU [12]. These rules are straightforward, but as this paper shows later, they often fail to produce suitable placements, and their effectiveness degrades further when GPU memory systems evolve across generations of GPU.

In this work, we introduce a new approach to address the data placement problem. We design the approach based on three principles.

First, the solution must have a good extensibility. GPU architecture changes rapidly, and every generation manifests some substantial changes in the memory system design. For a solution to have its lasting values, it must be easy to extend to cover a new memory system. Our solution features MSL (memory specification language), a carefully designed small specification language. MSL provides a simple, uniform way to specify a type of memory and its relations with other pieces of memory in a system. GPU memory has various special properties: Accesses to *global memory* could get coalesced, accesses to *texture memory* could come with a 2-D locality, accesses to *shared memory* could suffer from bank conflicts, accesses to *constant memory* could get broadcast, and so on. A challenge is how to allow simple but yet well-structured descriptions of these various properties such that they can be easily exploited by a data placement mechanism. Our solution is based on the following insight: All of those special properties are essentially about the conditions, under which, concurrent access requests are serialized. We introduce a

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.20

"serialization condition" field in the design of MSL, which allows the specification of all those special properties in logical expressions of a simple format. The design offers an underlying vehicle for data placement mechanisms to treat various types of memory in a single, systematic way. 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. This design, along with the compiler support explained next, allows code to be easily ported into a new GPU architecture with data placement automatically optimized (Section III).

Second, the solution should be input-adaptive. Different inputs to a program could differ in size and trigger different data access patterns, and hence demand different data placement. Since program inputs are not known until runtime, the data placement optimizer should be able to work on the fly, which entails two requirements. The first is to employ a highly efficient data placement engine with minimized runtime overhead. In our solution, we use an agile detection of data access patterns explained next, a lightweight memory performance model for fast assessing a data placement plan, and an open design that allows easy adoption of fast algorithms for searching for the best placement on the fly (Section IV). The second requirement is the ability to transform the original program such that it can work with an arbitrary data placement decided by the data placement engine at runtime. A typical GPU program does not meet the requirement because of its hardcoded data access statements that are valid only under a particular data placement. In our solution, we develop a source-to-source compiler named PORPLE-C, which transforms a GPU program into a *placement-agnostic form*. The form is equipped with some guarding statements such that executions of the program can automatically select the appropriate version of code to access data according to the current data placement. A complexity with this solution is a tension between the size of the generated code and the overhead of the guarding statements, which is addressed in our solution by a combination of coarse-grained and fine-grained versioning (Section V).

Third, the solution should have a good generality. Data placement is important for both regular and irregular GPU programs (our results show an even higher potential on irregular programs, detailed in Section VII.) A good solution to the data placement problem hence should be applicable to both kinds of programs. Here the main challenge is to find out the data access patterns of irregular programs, as they are typically not amenable for compiler analysis. Our solution is to employ a hybrid method. In particular, the PORPLE-C compiler tries to figure out data access patterns through static code analysis. When the analysis fails, it derives a function from the GPU kernel with its memory access patterns captured. The function contains some recording instructions to characterize memory access patterns. At runtime, the function runs on CPU, but only for a short period of time. This hybrid method avoids runtime overhead when possible and at the same time makes the solution broadly applicable. (Section VI)

Together, these techniques compose an extensible data placement engine named PORPLE (portable data placement engine.) PORPLE provides a general solution to GPU data placement with several appealing properties. 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 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 architectures, yielding up to 2.08X (1.59X on average) speedups on a set of regular and irregular GPU benchmarks, outperforming a rule-based method [12] significantly.

Overall, this work makes the following contributions:

- It presents the first general framework that supports cross-input, cross-architecture extensible optimizations of data placement for GPU.
- It introduces a simple yet flexible specification language that facilitates systematic description of various GPU memory systems.
- It proposes a way to produce placement-agnostic GPU programs through offline code staging and online adaptation.
- It describes a lightweight performance model for swift assessment of data placements and demonstrates its effectiveness for selecting suitable data placements for GPU programs.
- Through a comparison with manual and rule-based approaches, it empirically validates the significant benefits of PORPLE.

II. Overview of PORPLE

PORPLE enables cross-input, cross-architecture adaptive data placement through a combination of offline analysis and online adaptation.

As shown in Figure 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. These components are designed to equip PORPLE with a good extensibility, applicability, and runtime efficiency. Together they make it possible for PORPLE to cover a variety of memory, handle both regular and irregular programs, and optimize data placement on memory on the fly.

Even though PORPLE is possible to be extended to handle different types of data structures, it currently focuses on arrays, the most common data structure used in current GPU kernels. The methodology of PORPLE is independent of GPU programming models, but our implementation is on CUDA, which will be the base for the following description. We next explain each of the PORPLE components in further details.

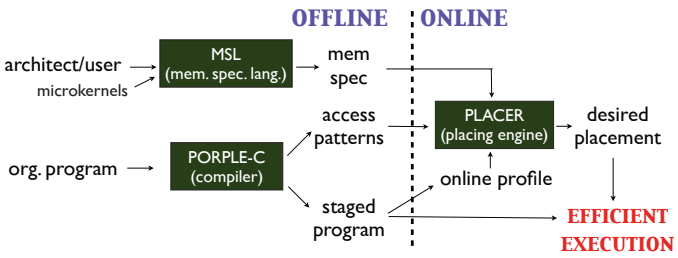

Fig. 1. High-level structure of PORPLE.

III. MSL: Memory Specification for Extensibility

An important feature of PORPLE is its capability to be easily extended to cover new memory systems. We achieve this feature by MSL. In this section, we first present the design of MSL, and then describe a high-level interface to enable easy creation of MSL specifications.

A. MSL

MSL is a small language designed to provide an interface for compilers to understand a memory system.

Figure 2 shows its keywords, operators, and syntax written in BackusNaur Form (BNF). An MSL specification contains one entry for processor and a list of entries for memory. We call each entry a spec in our discussion. The processor entry shows the composition of a die, a TPC (thread processing cluster), and an SM.

Each memory spec corresponds to one type of memory, indicating the name of the memory (started with letters) 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", if it is not "?", indicates that the spec entry is applicable only when the array dimensionality equals to the value of "dim". We will use an example to further explain it later. 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 blow 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 [11], 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 one.

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. Figure 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, which will be discussed in the next section.

B. Example

To better explain how MSL offers a flexible and systematic way to describe a memory system, we show part of the MSL specification of the Tesla M2075 GPU in Figure 4 as

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. The memory specification of Tesla M2075 in MSL.

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.

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. MSL expressions for the serialization conditions of various memory.

an example. We highlight two points. First, there are three special tokens in MSL: the question mark "?" indicating that the information is unavailable, the token "om" indicating that the information is omitted because it appears in some other entries, the token "na" indicating that the field is not applicable to the entry. For instance, the L2 spec has a "?" in its banks field meaning that the user is unclear about the number of banks in L2. PORPLE has some default value predefined for each field that allows the usage of "?" for unknowns (e.g., 1 for the concurrencyFactor field); PORPLE uses these default values for the unknown cases. The L2 spec has "om" in its upperLevels and lowerLevels fields. This is because the information is already provided in other specs. The L2 spec has "na" in its dim field, which claims that no dimension constraint applies to the L2 spec. In other words, the spec is applicable regardless of the dimensionality of the data to be accessed on L2.

Second, some memory can manifest different properties, depending on the dimensionality of the data array allocated on the memory. An example is texture memory. Its size limitation, block size, and serialization condition all depend on the dimensionality of the array. To accommodate such cases, an MSL spec has a field "dim", which specifies the dimensionality that the spec is about. As mentioned earlier, if it is "na", that spec applies regardless of the dimensionality. There can be multiple specs for one memory that have the same name and ID, but differ in the "dim" and other fields.

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. They are determined based on a prior study on GPU memory performance modeling [11]. To determine a kernel is compute or memory intensive, we measure the IPC during the profiling phase by checking performance counters (explained in Section VI). A kernel with IPC smaller than 2 is treated as memory-intensive, and compute-intensive otherwise.

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

C. GUI and Other Facilities

It is important to note that MSL specifications are not intended to be written by common programmers. The description of a type of hardware only needs to be written once—ideally by some architect or expert of the hardware. It can then be used by all programmers.

Architects or experts could choose to write the MSL specifications directly. But it could be error-prone. PORPLE provides a graphical user interface to enter the memory parameters and organizations with ease, from which, MSL specifications are automatically generated. The interface allows users to create a new memory component, drag it around, connect it with other components into a hierarchy, and fill out its latency, size, and other fields. During the MSL specification generation process, the generator checks for bugs in the specs (e.g., a name or ID used without defined, inconsistent hierarchy among the specs).

Architects or users could find out the parameters of a type of memory from architectural documentations, or use 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 [28], [24]. Users can add more of such microkernels into the library of PORPLE.

IV. Placer: Performance Modeling and Placement Search

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

A. 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 easily approximate the memory throughput, and hence assess the quality of the plan.

To that end, the Placer needs to determine the number of transactions needed by all the accesses to each array under a given data placement plan. It is simple to do 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 best data placement.

1) Reuse Distance Models. Reuse distance is a classical way to characterize data locality [4]. 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. Although this relation assumes a fully-associative cache, prior studies have shown that it is also effective for approximating the cache miss rates for set-associative cache [21], [30].

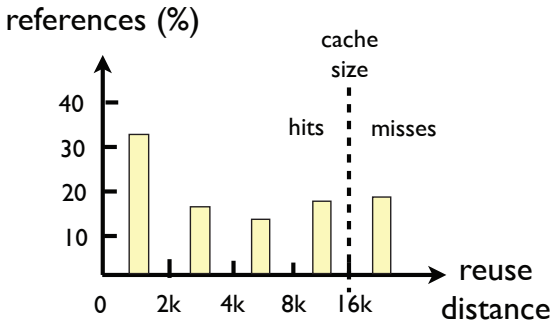

Fig. 5. Illustration of a reuse distance histogram and its relation with cache miss rate estimation.

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. Figure 5 shows an example, where, the second bar, for instance, shows that 17% of the references to the array have reuse distances in the range [2K, 4K). With the histogram, it is easy to estimate the cache miss rate for an arbitrary cache size: It is simply the sum of the heights of all the bars appearing on the right-hand side of the cache size as illustrated in Figure 5.

The histogram-based cache miss rate estimation comes handy especially for the runtime search for the best data placement by PORPLE. During the search, PORPLE needs to assess the quality of many placement plans, some of which have multiple arrays sharing a single cache. Following a common practice [17], upon the cache contention, the effects can be modeled as each array (say array i) gets a portion of the cache, the size of which is proportional to the array size. That is, the cache it gets equals $size_i / \sum_j size_j$, where, $size_j$ is the size of the jth array that share the cache. PORPLE then can immediately estimate the cache miss rates of the array by comparing that size with the bars in the reuse distance histogram. 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 [5], and reference traces [10] for irregular accesses. Construction from a trace has a near-linear time complexity [10]; 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 VI.

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.

There have been some GPU performance models studied before [2], [11], [20]. They mainly target on prediction accuracy, and are heavyweight in model construction and deployment. To meet the requirement of online usage, the performance model must be easy to build and agile to apply. PORPLE uses a lightweight model to approximate the memory throughput. The objective is to quickly determine the relative quality of different data placement plans, rather than giving out the most accurate performance prediction.

At the center of the model is a formula to compute the *memory throughput factor*:

$$1/\sum_{i \in allarrays} \sum_{j \in memHier(i)} N_{ij} * T_j * \alpha_j.$$

The inner summation estimates the total time that accesses to array i incur and the outer summation sums across all arrays. 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, and α_j is the concurrency factor explained in Section III, which takes into account that multiple memory transactions may be served concurrently (e.g., on global and texture memory). Together, the denominator tries to estimate the time taken by all memory transactions. The inverse hence reflects the memory throughput. A placement plan that maximizes the memory throughput factor is regarded as a best option.

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 surprisingly 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. As we will see in Section VII, the simple model used in PORPLE strikes a good tradeoff between complexity and usability.

B. Search for the Best

With the capability to determine the quality of an arbitrary data placement plan, the Placer just needs to enumerate all possible plans to find the best. There are many search algorithms that PORPLE could use, such as A*-search, simulated annealing, genetic algorithm, branch-and-bound algorithm, and so on. PORPLE has an open design by offering a simple interface; any search algorithm that is compatible with the interface can be easily plugged into PORPLE. The interface includes a list of IDs of software-managed memory, a list of array IDs, and a data structure for a data placement plan. PORPLE offers a built-in function (i.e., the memory performance model) that a search algorithm can directly invoke to assess the quality of a data placement plan. Users of PORPLE can configure it to use any search algorithm. All the aforementioned algorithms can find the placement plan that has the smallest total latency; they could differ in empirical computational complexity. However, in this work, we did not see a large difference in their efficiency, plausibly due to the limited number of arrays in the benchmarks. For kernels containing many arrays, approximated searching algorithms may help lower the search time. The reported results in Section VII are from the classic branch-and-bound algorithm. It 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.

V. PORPLE-C: Staging for Runtime Placement

When an array is put into different types of memory, the syntax needed to access the array is different. For instance, Figure 6 shows the syntax for accessing an element in array A in four scenarios. As shown in Figure 6 (a), 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 Figure 6 (b). The code is more substantially different when it comes to texture memory and shared memory. For texture memory, as Figure 6 (d) 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, which means that at runtime, the program is able to place data according to the suggestions by PORPLE, and at the same time, is able to run correctly

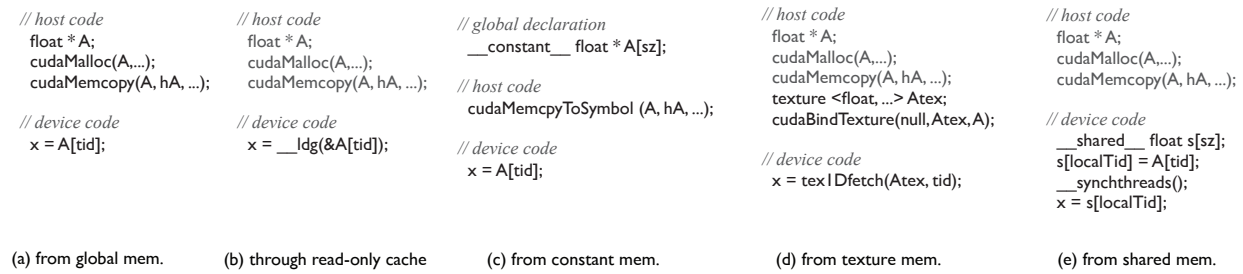

Fig. 6. Codelets in CUDA for accessing an element in an array A.

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 program through source-to-source translation. 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 Figure 7 (d). The figure shows the code generated by the compiler from a statement "A1[j]=A0[i]". Because the compiler is uncertain about where the array $A0$ 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 Figure 7 (b). 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 Figure 7 (d). 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 $A0tex$. Figure 7 (a) 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 Figure 7 (b).

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.

Figure 7 (c) 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 that 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., $sA0$ for $A0$) 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 Figure 7 (c) 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 Figure 7 (e).

4) Constant Memory. The final case is when the array is put into constant memory. For the extremely small size of constant memory, we allow at most one array to be put into it, which avoids introducing many auxiliary variables for referencing different arrays in constant memory. The compiler adds a global declaration for a constant buffer shown at the top of Figure 7 (a). Its size CSZ is set to the total size of constant memory. At runtime, when the "PORPLE_place" function decides to put an array into the buffer, it does it immediately as the call of "cudaMemcpyToSymbol" function in Figure 7 (b) shows. The changes to the kernel is just to replace the access to that array with the access to that constant buffer as "case CST" statement in Figure 7 (d) shows.

5) Compiler Implementation. The implementation of the compiler is based on Cetus [15], 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 7 (a));
> Step 4: customize PROPLE_place function accordingly (as illus. by Fig 7 (b));
> Step 5: insert code at the start and end of each kernel function and change data access statements (as illus. by Fig 7 (c,d,e)).

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 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 to utilize the new memory accordingly.

VI. Other Details

PORPLE does a lightweight on-line profiling, for two purposes. The first is to find out array sizes. Along with the data access patterns that PORPLE-C finds out, the array information serve for the Placer to search the best data placement. The second purpose 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, which keep 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 runtime, 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 overhead of the online profiling must be strictly controlled since it happens at runtime. PORPLE uses two techniques. First, the CPU function only performs the work of the first thread block. Second, if the kernel contains a loop, the CPU function only executes the first ten iterations. The overhead reduction techniques are 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 VII. Another technique to save the overhead is to discard computations that are irrelevant to data accesses, which is not implemented in PORPLE-C due to complexity of code slicing for implementing this technique.

Even with the above optimizations, the profiling time, in some cases, is still substantial compared to the running time of one kernel invocation. So the 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 the benefits from the many invocations of the optimized kernel.

VII. Evaluation

A. Methodology

We evaluate PORPLE on a diverse set of benchmarks shown in Table I. These benchmarks include all of the level-1 benchmarks from the SHOC benchmark suite [8] that come from various application domains. To further evaluate PORPLE with complicated memory access patterns, we add three benchmarks from the RODINIA benchmark suite [6] and three from CUDA SDK. The bottom five benchmarks in Table I 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

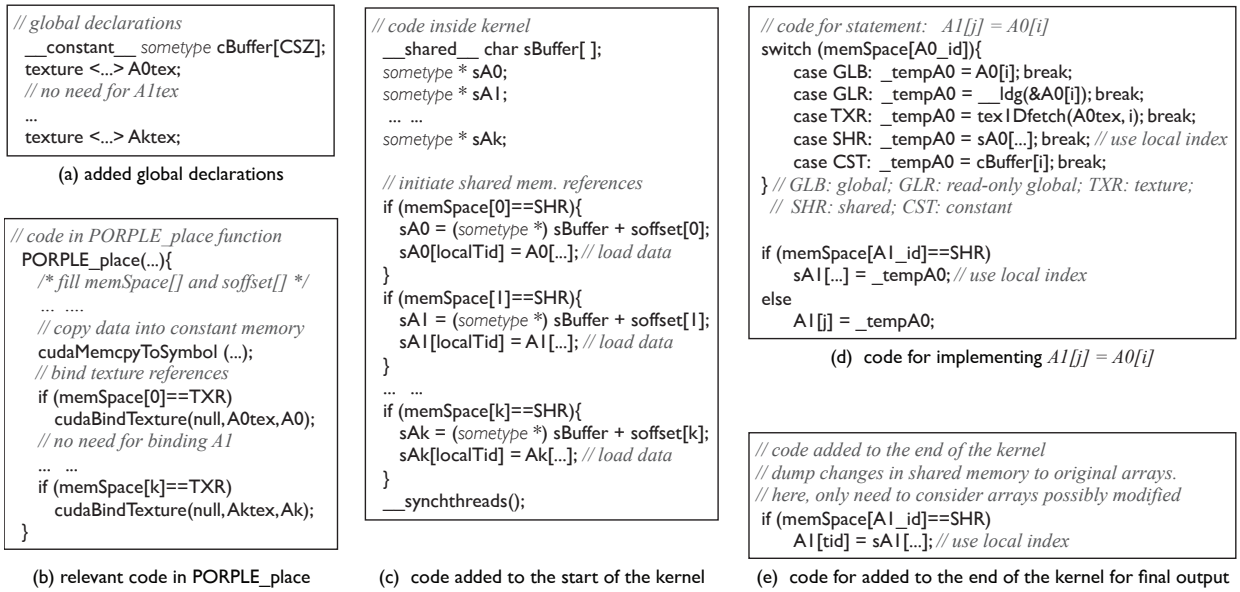

Fig. 7. Generated placement-agnostic code.

surrounding the GPU kernel call. The loop in *bfs* has a fixed number of iterations (100); while the numbers of iterations of the loops in the other four benchmarks are decided by the input argument of the benchmark. In our experiments, we use 100 for all of them. We focus on the optimization of the most important 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 our future study.

TABLE I
BENCHMARK DESCRIPTION.

Benchmark	Source	Description	Irregular
mm	SDK	dense matrix multiplication	N
convolution	SDK	signal filter	N
trans	SDK	matrix transpose	N
reduction	SHOC	reduction	N
fft	SHOC	fast Fourier transform	N
scan	SHOC	scan	N
sort	SHOC	radix sort	N
traid	SHOC	stream triad	N
kmeans	Rodinia	kmeans clustering	N
particlefilter	Rodinia	particle filter	Y
cfd	Rodinia	computational fluid	Y
md	SHOC	molecular dynamics	Y
spmv	SHOC	sparse matrix vector multi.	Y
bfs	SHOC	breadth-first search	Y

We evaluate PORPLE on three different machines with diverse GPU hardware and runtime environment shown in Table II. We choose different generations of GPU cards for the purpose of studying portability. The GPU cards have dramatically 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, and each SM has a user-controllable, read-only data cache, working as a L1 cache for global memory.

TABLE II
MACHINE DESCRIPTION.

Name	GPU card	OS	CUDA version
K20c	NVIDIA K20c	Linux-3.7	5.0
M2075	Tesla M2075	Linux-2.6	4.1
C1060	Tesla C1060	Linux-3.11	5.5

Since the specific features of different types of memory are proprietary information, we use a tool published by Wong [24] 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 III.

We compare PORPLE with the state-of-the-art memory selection algorithm published previously [12]. 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. We repeat each experiment for 10 times and calculate the average value of performance. 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.).

B. Results with Regular Benchmarks

Figure 8 shows the performance results for regular benchmarks on Tesla K20c. PORPLE provides on average 13% speedup, successfully exploiting almost all potential (i.e.,

TABLE III
MEMORY LATENCY DESCRIPTION. 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.

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

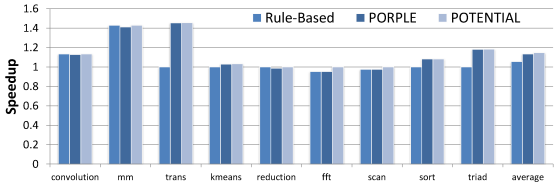

Fig. 8. Speedup of regular benchmarks on Tesla K20c.

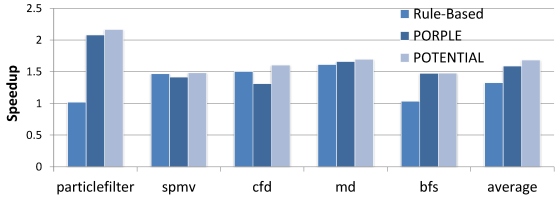

Fig. 9. Speedup of irregular benchmarks on Tesla K20c.

14%) from data array placement. The rule-based approach, however, only provides 5% performance improvement. We observe that for all benchmarks, except mm, trans and triad, the data placement strategies in the original programs are optimal or close to optimal, as the benchmark developers manually optimized the GPU kernels. Hence, there exists little potential (<10%) for further improvement. PORPLE and the rule-based approach both find the optimal placement strategy, which is almost the same as in the original programs.

The benchmarks mm, trans and triad show much larger speedup, ranging from 1.18X to 1.45X. PORPLE, as well as the rule-based approach, identifies the best placement strategy for mm. But PORPLE outperforms the rule-based approach by yielding 45% and 18% additional speedup for trans and triad respectively. 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 memory can be faster than global memory for those arrays, even if those arrays are linearly accessed. As a result, PORPLE gains significant benefits by placing all those read-only arrays in texture memory.

The runtime overhead for PORPLE is trivial (around 1%) for regular benchmarks, and hence they are not reported. As described in Section V, for these codes that are statically analyzable, 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.

We elide the results on the other two machines, as they are similar to those on K20c.

C. Results with Irregular Benchmarks

1) Speedup. Figure 9 shows the results for the irregular benchmarks on K20c. The complex memory access patterns of these benchmarks make any static analysis or manual optimization difficult. The original programs choose data placement strategies substantially inferior to the optimal. The optimal placement based on our profiling produces on average 1.68X and up to 2.17X (for particlefilter) speedup over the original ones. PORPLE provides 1.59X speedup on average over the original ones, only 9% less than the optimal, but 26% more than the rule-based approach.

We observe that the rule-based approach works well for some benchmarks, but fails in some others significantly. For example, it identifies the optimal placement strategy for cfd and md as PORPLE does, with slightly better performance than PORPLE because of PORPLE's runtime overhead. However, for particlefilter and bfs, the rule-based approach shows much less speedups compared to the optimal one, because it abuses texture memory and global memory for some arrays based on limited static analysis. In particular, for particlefilter, there is an array named CDF that can benefit better from constant memory than from texture memory, because the faster data caches in constant memory is helpful for good temporal locality associated with this array. 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 the size of CDF is unknown for static analysis. Also, the rule-based approach places too many arrays in texture memory and causes severe cache interferences which throttle the benefits on particlefilter.

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

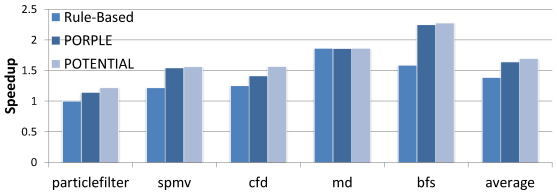

Fig. 10. Speedup of irregular benchmarks on Tesla M2075.

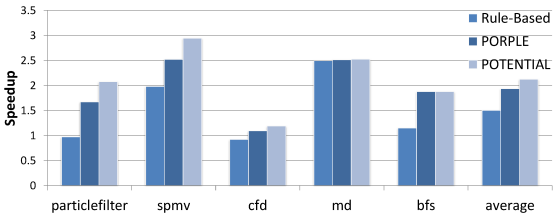

Fig. 11. Speedup of irregular benchmarks on Tesla C1060.

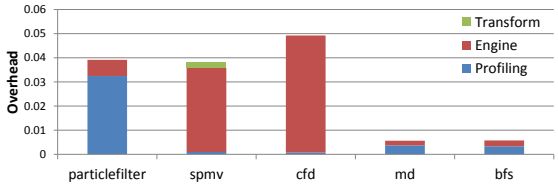

Fig. 12. The breakdown of overhead for irregular benchmarks on Tesla K20c.

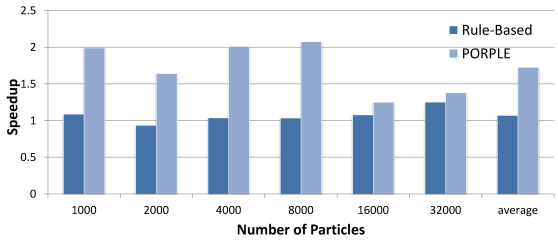

Fig. 13. Speedup across different inputs for particlefilter.

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 $levels1$ into the constant memory, achieving 1.44X speedups over the rule-based approach which uses only global memory for the arrays.

Figures 10 and 11 display the results on M2075 and C1060. PORPLE shows some performance gap from the optimal (6% and 19% less on M2075 and C1060 respectively), however it still performs much better than the rule-based approach (26% and 44% more on M2075 and C1060 respectively). The non-optimal results of PORPOLE are due to two reasons. First, PORPLE only profiles the memory accesses of the first thread block in order to minimize runtime overhead. However, the first thread block may not be a sufficient representative of other thread blocks, especially for irregular applications. 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. This fact is especially pronounced in the benchmark cfd on K20c. For this benchmark, PORPLE does not place all read-only arrays into texture memory because of concerns of cache interference, while the rule-based approach does not concern cache hierarchy, and places all arrays into texture memory, which leads to better performance than PORPLE.

2) Overhead Breakdown. Figure 12 reports runtime overhead of PORPLE on Tesla K20c. On average, PORPLE introduces 2.7% overhead, which is outweighed by the performance benefit as evidenced by the speedup results. The overhead can be decomposed into three parts: profiling, transform and the placement search (denoted as *engine* in the Figure 12). 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 breakdown varies dramatically across benchmarks. For particlefilter, the profiling on CPU is the main source of overhead, accounting for 3% of total execution time. For spmv and cfd, their overhead is dominated by the placement search, because they have a larger number of data arrays. For md and bfs, their overhead is very small (less than 1%), because their long execution times well amortize costs of PORPLE..

3) Portability. Table IV shows the placement decisions made by the rule-based approach and PORPLE. The rule-based approach is not portable, and always generates 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 input sensitivity of our method, we use six different inputs for particlefilter and spmv, and study their 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

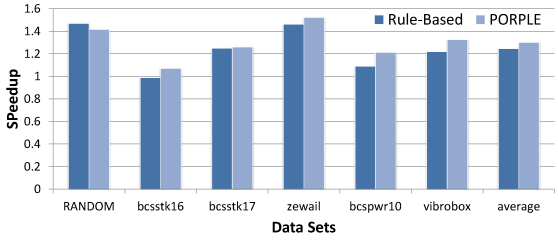

Fig. 14. Speedup across different inputs for spmv.

TABLE IV
PLACEMENT DECISIONS MADE BY PORPLE AND THE RULE-BASED APPROACH. 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.

	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

matrix database [9].

The results are shown in Figures 13 and 14. The figures show that PORPLE consistently outperforms the rule-based approach even if we use different inputs. In particular, the rule-based approach produces 7% and 24% average speedups for particlefilter and spmv respectively, while PORPLE produces 72% and 30% average speedups. The only outlier is the spmv with a special input named *random*. For this input, PORPLE performs worse than the rule-based approach. We attribute this to insufficient workload characterization by profiling the first thread block.

VIII. RELATED WORK

We categorize the related work into three classes.

1) Automatic Data Placement. The previous sections compared PORPLE with a rule-based placement approach designed by Jang et al. [12]. Ma and others [16] considered the optimal data placement on shared memory only. Wang and others [23] studied the energy and performance tradeoff for placing data on DRAM versus non-volatile memory.

Data placement is also an important problem for CPUs with heterogeneous memory architectures. Jevdjic et al. [13] designed a server CPU that has 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 [19], [18] involving phase change memory also chose hardware-managed data placement.

To our best knowledge, PORPLE is the first portable, extensible optimizer that systematically considers data placement on various types of GPU programs, enabled by its novel design of the memory specification language and the placement-agnostic code generator.

2) Performance Modeling.

Hong and others [11] propose a sophisticated GPU analytical performance model, which has many parameters and needs multiple runs of micro-benchmarks to determine them. The work by Zhang and Owens [28] also requires non-trivial executions of micro-benchmarks to establish quantitative model. Baghsorkhi et al. [1] proposes a compiler-based performance model, which considers very detailed micro-architectures, such as shared memory bank conflicts and warp divergence. All these approaches provide sophisticated modeling, but are costly and suit offline usage. The performance model in PORPLE is simple to build and use, efficient enough for online usage. It gives special consideration to cache and cache contention through the reuse distance model.

3) GPU Memory Optimization. GPU programs heavily rely on memory performance; memory optimization hence receive great attention [7], [3], [22], [14]. Yang and others [26] designed a source-to-source offline compiler to enhance memory coalescing or shared memory use. Zhang and others [27] focused on irregular memory references and proposed a pipelined online data reorganization engine to reduce memory access irregularity. Wu and others [25] formalize the problem of using data reorganization to minimize non-coalesced memory accesses, provide the first complexity analysis and propose several efficient reorganization algorithms. All these studies mainly focus on optimizing the memory access pattern rather than choosing the most suitable type of memory for data arrays. In this sense, PORPLE is complementary to them.

IX. CONCLUSION

Suitable data placement on GPUs produces substantial performance improvement, which, however, is hard to exploit by programmers due to the complicated memory system, the unpredictable run-time program behaviors, and the quickly evolving architectures. In this work, we propose the PORPLE framework to place data arrays on GPUs in a way that is transparent to programmers (in most cases), adaptive to inputs and extensible to new memory architectures. Our experiments on a diverse set of benchmarks and three different GPUs showed that PORPLE, by finding the optimal or near-optimal placement, consistently outperformed the manually optimized benchmarks and the state-of-the-art memory selection algorithm significantly.

ACKNOWLEDGMENT

The comments by the Micro'14 reviewers and Ayal Zaks helped enhance the presentation of the paper. This material is based upon work supported by DOE Early Career Award and the National Science Foundation (NSF) under Grant No. 1320796 and CAREER Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF. The work was partially performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 to the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

REFERENCES

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

[2] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. mei W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2010.

[3] 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 *ICS'08: Proceedings of the 22nd Annual International Conference on Supercomputing*, 2008, pp. 225–234.

[4] A. P. Batson and A. W. Madison, "Measurements of major locality phases in symbolic reference strings," in *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, Cambridge, MA, March 1976.

[5] G. C. Cascaval, "Compile-time performance prediction of scientific programs," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2000.

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

[7] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing memory access patterns for heterogeneous systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011, pp. 13:1–13:11.

[8] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *GPGPU*, 2010.

[9] 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.

[10] C. Ding and Y. Zhong, "Predicting whole-program locality with reuse distance analysis," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003, pp. 245–257.

[11] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *International Symposium on Computer Architecture*, 2009.

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

[13] 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 *ISCA*, 2013, pp. 404–415.

[14] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in gpus," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12, 2012.

[15] S. Lee, T. Johnson, and R. Eigenmann, "Cetus - an extensible compiler infrastructure for source-to-source transformation," in *In Proceedings of the 16th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003, pp. 539–553.

[16] W. Ma and G. Agrawal, "An integer programming framework for optimizing shared memory use on gpus," in *PACT*, 2010, pp. 553–554.

[17] S. Manegold, P. Boncz, and M. L. Kersten, "Generic Database Cost Models for Hierarchical Memory Systems," in *Proceedings of VLDB*, 2002, pp. 191–202.

[18] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, 2009, pp. 24–33.

[19] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11, 2011, pp. 85–95.

[20] J. Sim, A. Dasgupta, H. Kim, and R. W. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2012.

[21] A. J. Smith, "On the effectiveness of set associative page mapping and its applications in main memory management," in *Proceedings of the 2nd International Conference on Software Engineering*, 1976, pp. 286–292.

[22] 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 *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, 2010, pp. 513–522.

[23] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter, "Exploring hybrid memory for gpu energy efficiency through software-hardware co-design," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 93–102.

[24] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking." in *ISPASS*. IEEE Computer Society, 2010, pp. 235–246.

[25] 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 *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013.

[26] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10, 2010, pp. 86–97.

[27] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[28] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11, 2011, pp. 382–393.

[29] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3d stencil codes on gpu clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, 2012, pp. 155–164.

[30] Y. Zhong, S. G. Dropsho, and C. Ding, "Miss rate prediction across all program inputs," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, September 2003.