Coherence-Free Multiview: Enabling Reference-Discerning Data Placement on GPU

Guoyang Chen, Xipeng Shen
Computer Science Department, North Carolina State University
890 Oval Drive, Raleigh, NC, USA, 27695
{gchen11, xshen5}@ncsu.edu

ABSTRACT

A Graphic Processing Unit (GPU) system is typically equipped with many types of memory (e.g., global, constant, texture, shared, cache). Data placement determines what data are placed on which type of memory, essential for GPU memory performance. Prior optimizations of data placement always require a single view of a data object on memory, which limits the optimization effectiveness. In this work, we propose *coherence-free multiview*, an approach that allows multiple views of a single data object to co-exist on GPU memory during a GPU kernel execution. We demonstrate that under certain conditions, the multiple views can remain incoherent while facilitating enhanced data placement. We present a theorem and some compiler support to ensure the soundness of the usage of coherence-free multiview. We further develop *reference-discerning data placement*, a new way to enhance data placements on GPU. It enables more flexible data placements by using coherence-free multiview to leverage the slack in coherence requirement of some GPU programs. Experiments on three types of GPU systems show that, with less than 200KB space cost, the new data placement technique can provide a 1.6X average (up to 4.27X) speedup.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*optimization, compilers*

Keywords

GPGPU, Memory, Optimizations, Data Placement, Compiler, Runtime, Coherence

1. INTRODUCTION

A Graphic Processing Unit (GPU) system is typically equipped with many types of memory. On NVIDIA Tesla K20, for instance, there are eight types of memory: global memory, constant memory, texture memory, shared memory, various cache, and so on. They differ in size and other properties: Some are subject to bank conflicts or irregular accesses, some are for read-only data, some are good for 2-D or 3-D data locality, some are software controllable, some not. Such a complex memory system is largely motivated by the massive parallelism of GPU: By offering the large variety and flexibility, it could help meet the different needs of various GPU applications, and hence meet the relentless demands of the thousands of GPU cores for data.

However, in practice, the potential of such memory systems have been largely wasted. Although the many types of memory offer lots of flexibility, the complexity of the memory system has made it a daunting task for programmers to figure out the best data placement—that is, which memory should hold which data objects for a program execution. The support from commodity programming models and compilers is limited—just helping place read-only data ("const") into read-only cache [1]. As prior works have shown [2, 3], the best data placements are determined by many factors beyond the readl-only data property: the number of data objects, their sizes, access patterns, architectural features, program inputs, and so on. As a result, except some special purpose Graphics applications, GPU programs often fail to tap into the full power of the sophisticatedly designed GPU memory system. Prior studies have shown that better data placements can provide up to 3X speedups [2, 3] on GPU.

The importance of data placement has drawn some recent research interest. Jang and others have proposed a list of rules to guide users in placing data when writing a program [3]. Chen and others have developed a software framework named PORPLE to automatically place data during runtime [2].

Even though the efforts show some promising results, they are all subject to one constraint: A data object has only one single view throughout the execution of a GPU kernel. This constraint limits the exploitation of the full power of the various types of memory, especially when the GPU kernel contains multiple references to the same data object. The codelet in Figure 1, for example, contains three statements accessing a matrix A with a unique access pattern at each. At S1, all threads in a warp read the same element in the matrix; at S2, each thread writes to one element in the matrix; at S3, each thread gets the sum of a tile of four elements in A. The placements of A suiting the three statements differ: Putting it on the constant memory is the best at S1 for the broadcasting effects of constant memory (explained

ICS '16, June 01-June 03, 2016, Istanbul, Turkey
 ISBN 978-1-4503-4361-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2925426.2926277

```
// A is a matrix with width elements per row;
// tid: thead ID; wid: warp ID
S1:  … = A[wid];
     … …
S2:  A[tid+1] = …;
      … …
S3:  … = A[i]+A[i+1]+A[width+i]+A[width+i+1];
```

Figure 1: A codelet of a GPU kernel with multiple references to a single data object (synchronizations elided).

```
// Host side:                    // BFS_kernel at GPU:
S1:  iter ← 0;                   S7:  pick a node v for this warp to work with
S2:  do {                        S8:  if (levels[v] == iter) {// a node in the wave front
S3:    changed ← 0;              S9:    worklist[tid] ← k neighbors of node v
S4:    call BFS_kernel;          S10:   foreach w in worklist[tid] {
S5:    iter ++;                  S11:     if (levels[w] == UNSET) {
S6:  } while (changed)           S12:       levels[w] ← iter+1;
                                 S13:       changed ← 1;
                                 S14:  }}}
```

Figure 2: Pseudo-code for finding out the breath-first search (BFS) order (to store in array *levels*) of each node in a graph.

later), global memory suites S2 for the coalesced stores, and texture memory works the best for S3 for its 2-D data locality. However, all prior techniques [2, 3] would place A on the global memory, as they require only one single placement for a data object in a kernel while constant or texture memory is for read-only objects.

In this work, we address the fundamental limitation by proposing *coherence-free multiview*, an approach that allows multiple views of a single data object to co-exist on GPU memory during a GPU kernel execution *without coherence guarantees* among those views. It enables *reference-discerning data placement*, the first method to tailor the placement of a data object to the need of every individual reference to the object.

A key challenge in having more than one view of a data object is in maintaining coherence among the views. In the example in Figure 1, if we create three copies of A on three types of memory and use one for each statement, the update at S2 may fail to bring the other copies of A to date, possibly causing wrong computation results. This difficulty is a plausible reason for the single-view requirement in prior work on data placement.

An important insight underlying *coherence-free multiview* is that in many situations, a certain degree of slack exists in the requirement of memory coherence, exploitation of which can dramatically lower or completely remove the cost of reference-discerning data placement.

Figure 2 gives an example. The code is to find out the level of each node if a breath-first search (BFS) happens on a graph, and to store the levels into the array *levels*. Each call of the kernel visits all nodes in the entire graph and finds out the levels for some nodes. The kernel is called repeatedly until no changes happen to *levels*. Some slack of coherence requirement exists in the accesses to *levels*. Suppose that two threads i and j are both working on a node w (for it is a neighbor of two nodes) and $levels[w]$ is UNSET when the kernel starts. At a moment, thread j tries to read $levels[w]$ at statement S11, while at a previous moment, thread i has assigned $iter+1$ to $level[w]$ through statement S12. If these two threads work on two incoherent views of $levels$, thread j may see $levels[w]$ still equaling UNSET. This incoherence, however, does not hurt the program correctness, because the consequence is that thread j will execute S12, assigning $levels[w]$ with $iter+1$—which is exactly what thread i has assigned it. This slack of coherence requirement is due to the idempotent property of the assignment statement.

Slack of required coherence exists in many other algorithms, which often play some pivotal role in their domains. The aforementioned BFS algorithm [4], for instance, is a fundamental graph search algorithm that has been used to solve many graph theory problems, including Garbage Collection, testing a graph for bipartiteness, computing the maximum network flow, Cuthill-McKee mesh numbering, and so on. Other examples include Single-Source Shortest Path [5]—an algorithm crucial for navigation and path finding, B+ Tree Search [6]—a search algorithm widely used in file systems and database, Survey Propagation—an influential belief propagation algorithm for satisfiability testing and graph coloring, and the backbone of solvers of linear equation systems, Gaussian Elimination [6] and LU Decomposition [6].

In addition, coherence slack could come as side effects of code optimizations. An example is kernel fusion. When multiple GPU kernels are combined into one (e.g., to reduce kernel launching overhead), situations may appear for coherence slack to occur. Consider, for instance, a GPU kernel call of $K_A()$ followed by a call of $K_B()$. K_A reads an array X and kernel K_B writes to it. After they are fused into one, coherence slack on X (e.g., one view for reads, the other for writes) could become possible and beneficial to exploit (a concrete example in Section 5).

Coherence-free multiview is our proposed approach for leveraging the slack of coherence requirement for enhancing data placement in memory to further improve GPU program performance. The key question tackled by the technique is how to effectively translate the slack into data placement and program transformations that are **sound** and **beneficial**.

In the presentation of our solution, we first provide a formal definition of coherence-free multiview. We then introduce the *coherence-free theorem* and discuss how the theorem guarantees the soundness of the usage of coherence-free multiview to a GPU program (Section 3). Based on the theorem, we design a compiler module, which checks the conditions for coherence-free multiview to apply soundly. We integrate the compiler module into an existing software framework PORPLE, and further develop a set of runtime techniques to handle the new implications that coherence-free multiview imposes to runtime data placement. Together, these techniques lead to the first software framework for *reference-discerning data placement* (Section 4).

We test the framework on three generations of GPU with a set of irregular and regular GPU programs. The results indicate that the proposed techniques can safely leverage the slack in coherence requirement of a GPU program for enhanced data placements, leading to 1.6X average (up to 4.27X) speedup compared to the data placement selected by programmers in the original benchmarks (overhead counted). It is worth noting that in many cases, multiview is enabled by incoherent cache rather than multiple copies of a data object in memory. An example is the global memory view and texture memory view at a data object (explained later.)

As a result, the space cost of multiview is less than 200KB in our experiments. (Section 5)

Overall, this work makes several major contributions:

- To our best knowledge, this is the first proposal of leveraging slack in coherence requirement for enhancing data placement in memory.
- It introduces the concept of *coherence-free multiview* and conditions for its sound usage for data placement on GPU.
- It develops the technique of *reference-discerning data placement*, which, for the first time, allows tailoring data placement to each individual reference.
- It compares with start-of-the-art solutions, and demonstrates significant benefits and good cross-input adaptivity of the new technique on two generations of GPU.

2. BACKGROUND ON GPU MEMORY

This section provides some background on GPU and its memory systems. We use NVIDIA CUDA terminology [1] in the discussion.

Upon the launch of a GPU kernel, thousands of threads get created and start running on GPU concurrently. These threads are organized in a hierarchy: 32 threads form a warp and they execute in lockstep, a number of warps form a thread block, and all blocks form a grid.

To meet the large demands for data, a modern GPU typically consists of a number of types of memory. The memory system frequently changes across GPU generations. Take Tesla K20c (compute capability 3.5) as an example. There are four major types of software-manageable memory and five major types of cache on a K20c:

(1) Global memory: It is 4GB large off-chip memory. Coalesced accesses to it by a warp are more efficiently supported. Accesses to global memory go through a L2 cache. If the data are read-only, the accesses also go through a read-only cache (above L2) with a lower latency.

(2) Texture memory: It physically occupies the same piece of memory as global memory does, but offers a different view such that accesses to a data on texture memory go through L2 cache and a texture cache at the higher level (i.e., closer to the cores). It has some special support for 2-D or 3-D data locality, and is typically used for read-only data. Data on texture memory could be modified through *surface writes*, a special runtime API. However, the new value may not be visible to threads that access the data through texture memory API (because texture cache is not coherent with the memory).

(3) Constant memory: It is 64KB offline memory. It is for ready-only data. A special property is its broadcasting effect: When all threads in a warp access the same data element, the requests can be satisfied efficiently. But otherwise, the accesses get serialized. Accesses to constant memory go through L2 cache and constant cache.

(4) Shared memory: It is on-chip memory, much faster to access than other types of memory. The size can be configured as 16KB, 32KB and 48KB. It consists of 32 banks, allowing both read and write accesses. It is however subject to bank conflicts: When multiple threads access the same bank at the same time, the requests get serialized.

(5) Various cache: There are numerous types of hardware cache. They are not directly managed by software. All accesses to off-chip memory go through L2 cache; The size of L2 cache is 1536KB. Above L2, there are L1 cache, read-only cache, texture cache, and constant cache. Their latencies differ, but are all much shorter than the latency to L2 and off-chip memory. L1 cache on Tesla K20c is mainly for register spilling. Read-only cache shares the same physical cache with texture cache, used for accesses to read-only data on the global memory. Texture cache is for accesses to the texture memory. Constant cache is for accesses to the constant memory.

The complexity in memory system forms a major barrier for developing efficient GPU programs. We next describe how coherence-free multiview can help data placement to tap into the full power of such complex memory systems.

3. COHERENCE-FREE MULTIVIEW

Simply speaking, coherence-free multiview is to have multiple views co-existing on a memory system for a data object, while no coherences are guaranteed for these views throughout a set of computations. The lack of coherence guarantee makes the technique tricky to apply; care or some systematic safety check must be taken to ensure the correctness of the result.

This section first gives a formal definition of coherence-free multiview, and then discusses its connections with GPU data placement. After that, it analyzes the conditions needed for coherence-free multiview to apply correctly in various situations, and finally crystalizes the discussions into a simple theorem that can be used by compilers to ensure a sound application of the technique for GPU data placement.

3.1 Definition and Connections with Data Placement

We first define the term "view". In this paper, a *view* is in the perspective of data accesses by the processor. It refers to a way (or a path in the memory system) through which the processor accesses a data element. If a data element has two views, the processor could access the data element in two different ways: When there are two copies of the data on memory (e.g., one in global memory, the other in constant memory), accesses to them apparently differ in the paths. When there is only one copy on memory, it can still have multiple views. For a data element in the global memory for instance, the processor can access it as a normal access to the global memory, or as an access to texture memory (e.g., through "__ldg" intrinsic). The two accesses take different paths in the memory systems: The latter goes through texture cache while the former does not.

We define coherence-free multiview as follows:

DEFINITION 3.1. *A data object has a* coherence-free multiview *if it has more than one view in the memory system, which are actively used but not guaranteed coherent throughout a set of computations.*

We emphasize that two *views* could be two copies of the data object in different pieces of memory (e.g., one on constant memory, one on texture memory), but they could also be two incoherent views of a single copy of the data object on memory. An example is a data object on texture memory on GPU. As the previous section has described, the object can

also be treated as an object on the global memory, and get modified through some special surface API. Accesses to texture memory go through texture cache, but accesses through surface API do not. As a result, incoherence could exist among the two views: Threads read a data object through texture API may not see the up-to-date value of the object on the memory if the object has been modified through surface API.

It is worth noting that the definition of coherence-free multiview requires that more than one view of the object is active during the set of computation. It excludes the cases where incoherence exists among multiple views but only one of the views is active for computation. An example is the usage of GPU on-chip shared memory, which is often used as a stage for accelerating global memory accesses. It is a common practice that at an early part of a kernel, some data are copied from the global memory into the shared memory, and then the kernel just works with (reads and writes) that copy of the data. Even though during the computations, the copy in the shared memory may be incoherent with the copy in the global memory of the data, it is not a coherence-free multiview because only the copy in the shared memory is actively used through the computation. The incoherence in that case does not need special treatment since it creates no safety concern. Similarly, L2 cache and global memory could each hold a copy of a data; but because L2 cache is made coherent by hardware, it is not coherence-free multiview either.

Connections with data placement. Coherence-free multiview helps overcome a fundamental limitation of prior data placement techniques, bringing new opportunities in three aspects.

(1) Data object aspect. It helps relax some constraints that prior solutions have imposed on a data object. Each type of memory has some requirements on compatible data access patterns. For instance, constant memory on GPU is designed for accommodating read-only data. If a data object is both read and written in a kernel, traditional data placement techniques would rule out the usage of constant memory for that data object. With multiview, constant memory could still benefit that data object. The object can, for instance, have two copies with one in the constant memory and the other in the global memory. Reads to that data object can then go to the constant memory, while writes go to the global memory. Another example is texture memory. The guideline used by all prior data placement techniques is that only read-only data can be put into texture memory because texture cache is not coherent with memory. With coherence multiview, as long as coherence-free multiview can be applied to a data object safely, the data can be put onto texture memory.

(2) Data reference aspect. Multiview enables reference-discerning data placement. A kernel may contain multiple references to the same data object. Under traditional data placement, no matter how different their access patterns are, they must access the data object in the same way. With coherence-free multiview, it becomes more flexible. Each reference instruction may access the data object through the view best fitting its access pattern.

(3) Runtime overhead aspect. Creating multiviews for an object may not be difficult. The fundamental reason why no prior data placement technique has done it is the runtime overhead and difficulty to maintain coherence among those views. An important property of coherence-free multiview is that coherence among views is not necessary for maintaining the soundness of the execution, which eliminates the need and overhead for runtime coherence maintenance, making multiviews feasible to serve as a beneficial technique.

3.2 Coherence-Free Theorem

To materialize the new opportunities, a key challenge is to determine when coherence-free multiview can be applied safely. Before answering the question, we first define *coherence-free multiview transformation* formally as follows:

Definition 3.2. *Given a kernel that has a single view for an object B,* coherence-free multiview transformation *transforms the kernel with only the following effects: In the transformed kernel, there are multiple identical views created for object B before the kernel accesses B, and multiple of the views could be actively accessed by the kernel while coherence is not guaranteed among them. All writes to B go to only a single view.*

We propose the following theorem, using which, a compiler can easily ensure the soundness of the application of coherence-free multiview transformation.

Theorem 3.3. **Coherence-Free Theorem:** *Coherence-free multiview can be safely applied to an object if the GPU kernel meets both conditions: (1) There are no intra-warp true dependence on the object; (2) If there are inter-warp true dependences on the object, there are no synchronizations dictating the order between reads and writes to that object.*

In the theorem, a true dependence means that there are reads and writes to the same data object and some of the reads happen after some of the writes. If these reads are by a warp different from the warp conducting those writes, the dependence is called an *inter-warp true dependence*, otherwise, it is an *intra-warp true dependence*. Synchronizations include barriers for a group of threads and atomic memory operations.

To prove the correctness of the theorem, we categorize all possible scenarios into four classes based on the kinds of data dependences existing on the data object in question, and examine each of them as follows.

(1) Read-only objects. When the object is read-only in a kernel, the conditions of the theorem apparently hold. In this situation, lack of coherence among the multiple views of the object is obviously not an issue. Even if different instructions read from different views of the object, they read the same value as those views are identical at the creation time.

Yet, having multiviews for such objects on GPU can be beneficial. For instance, consider a kernel that has two statements reading array A; at the first, all threads in a warp read the same element, while at the second, they read different elements. We may create a copy of A on both the constant memory and the texture memory, and let the first access use the first view to get the broadcasting benefit, and the second access use the second view to get the locality benefit. The program could run faster if the copy-creation overhead is smaller than the benefits.

(2) Write-only objects. For write-only objects, having multiple views but writing only one of them is no better

than having only one view. It may be tempting to think that it could have some efficiency benefit to let different writes (of different access patterns) operate on different views of an object. However, for GPU, there are no such needs in practice. Write-only objects can be placed simply onto the global memory; placing them onto other types of memory is either not allowed or unbeneficial. An exception is putting it onto shared memory, which does not create *coherence-free* mutliviews as discussed earlier in this section.

(3) Read-before-write objects. An object is a read-before-write object if the object is both read and written in a kernel, and throughout the execution of the kernel, *all* reads to it happen before any write to it. There are apparently no intra-warp or inter-warp true dependences on that object. The correctness of coherence-free multiview can be seen as follows. When coherence-free multiviews are applied to the object, the reads are safe as long as the views are identical at the creation time. The correctness of writes are similar to that in the write-only scenario: Since all writes to the object still go to a single view, it is equivalent to the writes in the original kernel.

(4) Other scenarios. In other scenarios, the object is both read and written in the kernel, and some reads to that object may happen after some writes to it. These reads and writes can form true dependences. Incoherence among views could cause a violation to the true dependence and hence incorrect execution results. For such objects in a GPU kernel, coherence-free multiviews are still possible to be applied safely, but care must be taken. We examine each case of this scenario as follows.

(a) *There are intra-warp true dependences on that object.* In GPU, all threads in a warp execute in lockstep. If the reads and writes of the true dependence happen to operate on different views of the data object, the reads would get the obsolete values of the data object. Coherence-free multiview is generally unsafe to apply to such objects. The first condition in the coherence-free theorem excludes this case.

(b) *There are not intra-warp but inter-warp true dependences.* We separate such cases into two classes. In the first class, the kernel contains some synchronization statements (e.g.,__synchthreads or atomic operations) to (partially) dictate a certain order of the reads and writes to the data object by different thread warps, and these synchronizations are necessary for the kernel to run correctly. In the second class, the kernel either does not contain such statements, or those statements could be safely removed without affecting the correctness of the program. For the first class, coherence-free multiview could cause reads of obsolete values and is not applicable in general. It is excluded by the second condition in the coherence-free theorem.

The second class meets the conditions required by the coherence-free theorem. Coherence-free multiview can be applied safely. The reason is that the lack of needs for synchronizations entails that the order of the reads and the writes by different warps is not essential to the correctness of the program. In another word, the kernel contains races among reads and writes by different warps; however, the fact that no synchronization are necessary for that object indicates that the races do not affect the correctness of the program.

The BFS example in Section 1 already illustrates such a case for the idempotence of its writes. Figure 3 illustrates another example SSSP, in which, data race is not a problem due to the monotonic property of the writed values. In this example, the program tries to compute the length of the shortest path from a node *src* to all other nodes in a graph. The GPU kernel updates the length of the path from *src* to some nodes. The program repeatedly calls the GPU kernel until no nodes' path length gets changed. The read to "length[d]" at line S9 and the write to "length[d]" at line S10 form a race condition. So if we put the lengths associated with the nodes onto texture memory, read from it and use surface API for the write, the read by thread i could get an obsolete value of "length[d]" if another thread has just modified it for the lack of memory coherence. However, the new value of "length[d]" will be seen in the next call to the kernel. From S9 and S10, it is easy to see that the value of a length can never increase across the iterative computations, which, as proved by prior work [7], guarantees that the program terminates at a single fixed state, despite all the races.

```
// Host side:                          // SSSP_kernel at GPU:
S1:  set weight of all                 S5:  worklist[tid] ← k nodes
     nodes to ∞                        S6:  foreach s in worklist[tid] {
S1:  src.weight=0;                     S7:      foreach edge e coming out of s {
S1:  do {                              S8:          d = e.destination;
S2:     changed ← 0;                   S9:          if (length[d] > length[s] + e.length) {
S3:     call SSSP_kernel;              S10:             length[d] > length[s] + e.length;
S4:  } while (changed)                 S11:             changed ← 1;
                                       S14: }}}
```

Figure 3: Pseudo-code of SSSP computing the shortest path from a node *src* to all other nodes in a graph.

The examples demonstrate the existence of coherence requirement slack in some algorithms. For implementations of such algorithms with unnecessary synchronizations, users can first use some existing tools [7] to remove the synchronizations before applying data placement optimizations.

Overall, the *coherence-free theorem* provides a simple way for compilers to safely apply coherence-free multiview for data placement. The compiler only needs to examine the two conditions indicated in the theorem rather than enumerate all possible scenarios. When threads of a warp may diverge on a condition statement, the execution order of the two branches have some uncertainty. The compiler needs to consider both possible execution orders when examining data dependences. Upon a failure of the conditions in either case, the data object is excluded from the multiview optimization. We detail the implementation of the compiler support next.

4. REFERENCE-DISCERNING PLACEMENT

This section presents how a compiler module helps materialize coherence-free multiview, and how the enabled reference-discerning data placement is implemented. The implementations are based on an existing data placement framework named PORPLE. We review it first.

4.1 Background on PORPLE

PORPLE is a framework recently developed [2] to enhance data placement for GPU programs automatically. It emphasizes portability. It decides the best data placement and places data at runtime, such that the program can run with a placement suitable for the current platform and current

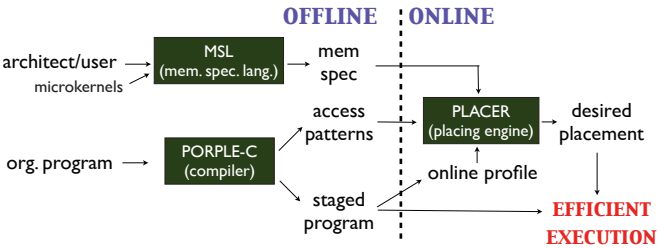

Figure 4: High-level structure of PORPLE.

inputs.

As shown in Figure 4, PORPLE consists of three major components: MSL, PROPLE-C compiler, and runtime data Placer. Architects are expected to write, in a specific language MSL, a specification about the memory system of the GPU of interest, covering the attributes of the various types of memory in a GPU model (size, hierarchy, sharing scope, latency, serialization conditions, etc.). The *PORPLE-C compiler* is a source-to-source compiler. It finds out the data access patterns from the given GPU program (in CUDA), and transforms the program into a data placement-agnostic form (still in CUDA) such that the program can run correctly regardless where the data are placed in memory. The runtime engine *Placer* complements the static analysis results with some information (e.g., data size, reuse distances of irregular accesses) collected through a lightweight runtime profiler. *Placer* then consumes those program-level properties and the MSL specifications to find the best data placements, and places data accordingly on the fly.

Take *BFS* as an example. Its GPU kernel contains accesses to three arrays, one of them (*edgeArray*) regularly accessed, two of them (*levels, edgeArrayAux*) irregularly accessed (i.e., not amenable to static analysis). The compiler finds out the access patterns of the regular ones. When the GPU program starts running, before the kernel gets invoked, the lightweight profiler (runs on CPU) obtains the size of the arrays as well as a short access trace to the irregularly accessed arrays (for only a small number of accesses.) *Placer* integrates the info together and uses heuristic search to find the best data placement. During the search, when examining one possible placement, *Placer* assesses the expected memory performance of the GPU kernel if that placement is used. The assessment is based on a performance model that *Placer* customizes to the current GPU memory system based on the MSL specification of the GPU. The performance model takes into consideration of the contention of data accesses on the shared cache and memory bandwidth on GPU. After the search, *Placer* gets the best placement plan it finds. The rest of the execution of the program—thanks to its placement-agnostic form—places the data according to that plan on GPU and benefits the enhanced memory performance brought by the new placement.

PORPLE focuses on array data structure for its importance in GPU kernels. Our implementation inherits such a focus. PORPLE does not support coherence-free multiview.

4.2 Compiler Support

Based on the source-to-source compiler in PROPLE, we develop the compiler support for coherence-free multiview. The support are mainly in two aspects.

Applicability Check. First, it checks the applicability of coherence-free multiview to each of the arrays used in the GPU kernel of interest. The check examines the conditions mentioned in the coherence-free theorem. It consists of three steps. First, it removes all written-only arrays from consideration. Second, it uses standard data dependence analysis to check whether there are intra-warp true dependences. If so, it removes that array from further consideration. Finally, it checks whether an array carries inter-warp true dependences. If so, it tries to determine whether there are synchronizations in the code that dictate the order of accesses related with the inter-warp dependences. If not, it includes that array into a *candidate set*. Otherwise, it excludes the array from the consideration, and at the same time, provides users a feedback that the synchronizations prevent coherence-free multiview from being applied. Users can then invoke tools [7] to remove unnecessary synchronizations, and rerun the compiler module. Integration of such tools into our compiler module is left to future work.

Per-Reference Pattern Collection. Unlike the data access patterns in PORPLE, our compiler module characterizes data access patterns at the reference level rather than object level. The compiler module records the access patterns of each reference instruction on every data object in the *candidate set*. The pattern includes whether it is read or write, and the access stride across threads, warps, and thread blocks. When the access patterns of a candidate array are difficult to analyze by the compiler (e.g., dynamic irregular accesses), an extended version of the lightweight runtime profiling in PORPLE will be used for them. In the extended version, the instrumentation by the compiler has been modified such that the profiler collects access patterns for each statement that accesses those candidate arrays. These statement-level access patterns will be later grouped at runtime to guide data placement as discussed next.

4.3 Runtime Support

As mentioned, for adapting to program inputs, PORPLE finds the suitable data placement plan at runtime and place data accordingly. Coherence-free multiview gives some new requirements to the runtime support, which are met by several newly developed features.

Multiview Locality Inference. The first new implication is on data locality inference. Section 4.1 has mentioned that an important step in runtime data placement is to assess the quality of each placement plan. For a data object whose accesses go through cache, the assessment must estimate the cache performance. The estimation requires the information on reuses of the data object; if the cache is shared by accesses to multiple objects, the estimation must also consider cache contentions among the objects.

The solution in PORPLE is to build up a reuse distance histogram for every data object, and then estimate cache hit rates accordingly: Given the cache size, accesses whose reuse distances are smaller than the cache size are considered as cache hits. It assumes full associativity, but in practice, also works reasonably well for set-associative cache. If k objects share the cache, PORPLE regards $1/k$ as the fraction of cache effectively used by an object.

The method however cannot apply in the presence of coherence-free multiview. Consider a kernel that contains accesses to

k arrays, A_i $(i = 1, 2, \cdots, k)$, and for each array, there are r instructions accessing it in the kernel. If any combination of the r instructions can access the same view of an array, there would be totally 2^r possible combinations of instructions for just one array. If we build a reuse histogram for each combination, we would need to build $k * (2^r - 1)$ reuse histograms for that kernel, which is generally infeasible given that PORPLE requires the construction and cache performance estimation to happen at runtime (since some information like data size is not known until then).

Our solution is *spec-guided pattern grouping*. The basic idea is to combine the locality analysis with the memory specification that is already provided to PORPLE. One insight is that in light of the properties and constraints of the various memory in the system, many of the possible combinations mentioned earlier are either illegal or apparently unprofitable. For instance, a write instruction cannot access a view on constant memory, and a read instruction with nonezero stride across threads does not suite constant memory. Based on the properties of common types of GPU memory, we derive the following several classes to categorize references: read-zero, read-regular, read-irregular, write-regular, write-irregular; the names indicate whether the access is a read or write, and whether the access has a regular stride; "read-zero" is a class whose stride across threads in a warp is 0, which is designed particularly for the broadcasting requirement of constant memory. At runtime, the instructions accessing an array are classified into those several classes. Instructions in the same category will definitely access the same view of that array. Different categories could access the same view, but we now only need to consider at most 31 combinations of the five categories, regardless of how many instructions access that array. A reuse histogram is built for each combination. In all kernels we have seen, references to an object fall into at most two of the classes—only three histograms are needed for an object.

Multiview-Conscious Data Placement. Coherence-free multiview also imposes some new implications to the search for the best data placement. In the original PORPLE, each placement plan specifies only one location for an object. With multiview, a plan needs to specify which view of an object each reference shall use. The pattern-based grouping mentioned earlier may reduce the requirement to which view each *group* of references shall use. But still, one object may now have more than one view specified in one placement plan, which entails a new complexity: Some of the views of an object can correspond to a single physical copy of the object, while some others have to correspond to different copies. For instance, if the two views for an array is "an array on the global memory" and "an array on the texture memory", they can refer to the same copy on the device memory. The difference is the way in which the array is accessed in the program code. On the other hand, if the second view is "an array on the constant memory", they must refer to two different copies.

There are two implications. First, for the cases when multiple physical copies need to be created, the quality assessment of a placement plan should consider the copy overhead. The overhead can be easily estimated with the data size and the MSL memory specifications. Second, the memory specification should include constructs to indicate the scenarios where two views refer to the same copy. To that end, we extend MSL with a new construct: "alias memID*" for specifying aliased views. PORPLE runtime counts data copy overhead only once for such aliased views when assessing placement plans.

Table 1: Machine Description.

Name	GPU card	Processor	CUDA
M2075	Tesla M2075	Intel Xeon X5672	5.5
K20c	NVIDIA K20c	Intel Xeon E5-1607v2	6.5
GTX980	GeForce GTX980	Intel Xeon E5-1607v2	7.0

5. EVALUATION

Coherence-free multiple views give new opportunities for enhancing data placements on GPU. This section examines the benefits through comparisons with the state of the art.

5.1 Methodology

We collect a set of important algorithms that have slack in coherence requirement, including the Breadth-First Search (BFS) [4], the Survey Propagation (SP) [5], the Single-Source Shortest Paths (SSSP) [5], the B+ Tree Search (b+tree) [6], the Gaussian Elimination (gaussian) [6], and the LU Decomposition (lud) [6]. As Section 1 mentions, each of these algorithms is pivotal to some domains. Speeding them up can have some substantial impact. When choosing the set, we try to include only those that already have GPU implementations available to public (SHOC [4], LonestarGPU [5], or Rodinia [6] suites). That allows a direct objective comparison. In the implementations, the first four algorithms have irregular (indirect) array accesses, while the other two have only regular accesses. Meanwhile, we include all the programs in level-1 SHOC benchmark suite [4] that do not have slack of coherence; they allow us to see whether the coherence-free multiview transformation causes negative effects on such programs. In addition, we take the MVI benchmark from Polybench [8] to demonstrate the cases where opportunities for coherence-free multiview come as side effects of code transformations (detailed in Section 5.3).

We implement our technique on the PORPLE framework. It allows a head-to-head comparison with the original PORPLE. Because PORPLE currently supports only CUDA, our experiments run on NVDIA GPUs. The technique and idea however are not limited to such platforms; other platforms (including OpenCL) may expose even more opportunities as discussed in Section 6. All programs, including both the original version and the version transformed by our source-to-source compiler, are compiled into native code through the latest NVIDIA NVCC compiler.

To examine the portability of the automatic data placement, we test on three different machines as shown in Table 1. M2075 has global, texture, constant and shared memory. Accesses to the first three all go through L2 cache, but are cached in their specific first-level cache (L1, texture, constant cache). Shared memory and global L1 cache share a 64KB on-chip physical memory, which can be configured to 48KB shared memory with 16KB L1 or the other way around. K20c additionally allows half-half split between L1 and shared memory. Its L1 is for register spilling rather than caching global loads. It adds a read-only cache, which in fact shares the space with texture cache. GTX980 features a unified L1/texture cache. However, multiviews of a data item can still exist in the unified cache: A data item

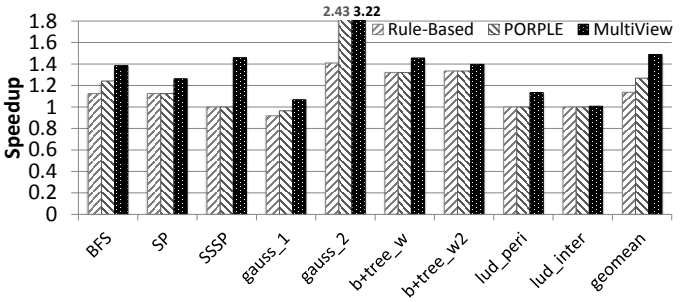

Figure 5: Speedup of Benchmarks on GeForce GTX980.

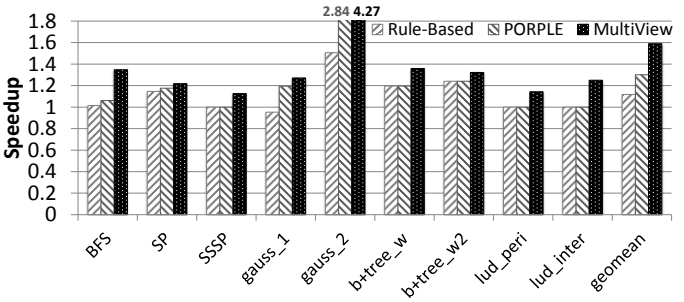

Figure 6: Speedup of Benchmarks on Tesla K20c.

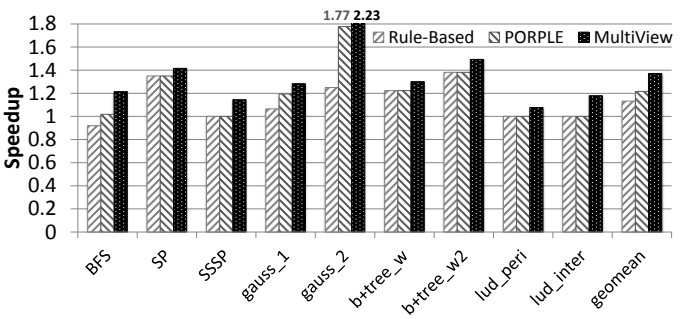

Figure 7: Speedup of Benchmarks on Tesla M2075.

may have two copies in the cache if it is loaded once through the traditional L1 cache (-Xptxas -dlcm=ca) and the other time through the texture cache (__ldg). Multiview is hence still applicable on this latest GPU.

We compare our method with three alternatives. The first is the placement determined by the programmers of the original benchmarks, the second is by a rule-based method [3] which places data based on several rules on some properties of the kernel code, and the third is by PORPLE [2]. The original PORPLE does not support surface writes; we added the support for a fair comparison.

All runtime overhead of the data placement framework has been counted in the time measurement, including the time to create all the extra data copies that multiview transformations could need, to profile data accesses (for irregular references), and to search for the best placements. We repeat each experiment for 10 times. As no significant variances are observed, the average is reported. The speedup baseline is the performance of the original benchmarks.

5.2 Performance Results

Overall Results.

On the level-1 benchmarks in SHOC that do not meet the conditions of coherence-free multiview, our method avoids applying multiview transformations to them. The performance remains the same as what PORPLE provides, demonstrating that the Multiview method does not cause negative effects on such programs.

We next focus on the results on the six algorithms that meet the multiview conditions. Three programs, *gaussian, b+tree, and lud*, each have two kernels meeting the conditions, shown separately in the performance graph. Figures 5, 6, and 7 report the speedups on the three generations of GPUs. Three programs, *gaussian, b+tree, and lud*, each have two kernels meeting the conditions, shown separately in the performance graph. On the latest architecture GTX980, as Figure 5 shows, the rule-based approach provides average 1.14X speedup, PORPLE provides average 1.27X (up to 2.43X) speedup, while Multiview provides average 1.49X (up to 3.22X) speedup. On the other two generations of GPUs, the degrees of speedups differ slightly, but Multiview consistently outperform the alternatives significantly, yielding 1.59X and 1.37X average speedups on Tesla K20c and Tesla M2075 respectively.

Multiview incurs space cost only when extra data copies are created in either constant or shared memory; as aforementioned, texture memory and global memory are two views of the same data copy. Therefore, the space cost on the three GPUs can be no larger than the sum of the constant memory, which is 0.015%, 0.014%, and 0.013% of the global memory size on the GTX980, K20c, and M2075 respectively. In our experiments, the actual space cost is at most 200KB.

The reported speedups already count all runtime overhead. Figure 8 gives a breakdown of the overhead on the irregular programs. The overhead consists of the time to create extra data copies ("Multiview_copy"), runtime profiling for data locality analysis ("Profiling"), model-based search for good placements ("Engine"), and execution of some conditional checks inserted by the code transformation ("Transform"). The first part is specific to multiview; it is trivial given how small the extra data copies are. The other parts have some noticable overhead; however, they are much smaller than the benefits of the optimizations. The runtime overhead of the optimization on regular benchmarks is even smaller since they need no runtime profiling.

Details. We now give some detailed discussion on each kernel to see how the multiview placement outperforms the

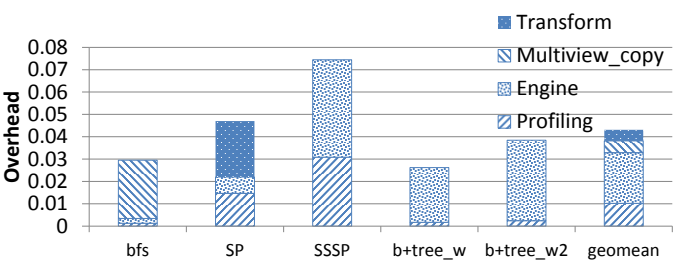

Figure 8: Runtime overhead on irregular benchmarks on Tesla K20c.

Table 2: Data Placement Strategy for BFS on K20c.
(#R/W-A[index]: the #'th reference of array A is read or write. The order of references in a statement is from right to left.)

Method	**1R-levels**$[v]$	**1R-edgeArray**$[v]$	**2R-edgeArray**$[v+1]$	**1R-edgeArrayAux**$[i+nbr\_off]$	**2R-levels**$[v]$	**3W-levels** $[v]$
Rule-based	Global	Texture	Texture	Texture	Global	Global
PORPLE	Global	Texture	Texture	Global	Global	Global
Multiview	Constant	Texture	Texture	Global	Global	Global

Table 3: Bandwidth Usage for Gaussian_fan2

Versions	**Texture cache**	**L2 cache**	**Device Memory**
Original	0	151.4GB/s	52.3GB/s(low)
Rule-based	20.3GB/s	164.5GB/s	63GB/s
PORPLE	0	218.5GB/s	81.4GB/s
Multiview	188.4GB/s	270.8GB/s	125.6GB/s(high)

other methods. We concentrate our discussions on K20c.

(1) Benchmark BFS. For BFS, the rule-based method only provides 1% performance improvement and PORPLE provides a better solution with 6% performance improvement. Based on PORPLE framework, the multiview method provides 35% performance improvement by considering benefits from all views of one array. Table 2 shows the data placement strategies generated by three different methods. The rule-based method places array *edgeArray, edgeArrayAux* on texture memory and array *levels* on global memory. PORPLE places array *edgeArray* on texture memory to benefit from the texture cache, and puts arrays *edgeArrayAux and levels* on global memory. Even though there are many data reuses at the first reference to *levels* that can better benefit from memories other than the global memory, PORPLE does not allow it to be put onto texture memory or constant memory because it is not read-only.

As explained in the previous sections, our method is able to recognize that it is safe for copies of *levels* to be incoherent. Accordingly, our multiview method creates another copy array *levels1* on constant memory for the first reference of *levels* (S8 in Figure 2) to read from. It brings lots of benefits by leveraging the broadcasting effects of constant memory since the threads in one warp always read the same address. The second reference of *levels* read from the copy on the global memory, the access pattern of which does not benefit from constant memory. The last reference of *levels* writes to global memory. After the kernel finishes, data in array *levels* is copied to *levels1* to update its content. The final speedup has already counted the overhead of copying data to constant memory.

We also experiment with a larger input graph (100000 nodes), whose array *levels* cannot fit into constant memory anymore. In this case, our optimizer creates a view of array *levels* on global and texture memory respectively, achieving 1.19X speedup.

(2) Benchmark SSSP. Similar to BFS, for benchmark SSSP, in each invocation, the kernel will go over the whole graph and try to update the value of each node. The original program uses an atomic operation when updating the values of each node to avoid data races, which affects the performance. The atomic-free method [7] can help remove the atomic operations by allowing data race for array *length*, as the kernel invocations are topology-driven and the value of each node has a monotocity as mentioned in the previous section.

After the removal of the atomic operations, our method creates multiple views for array *length*. To balance the texture bandwidth usage and global bandwidth usage, the placement strategy made by Multiview is that the first read of array *length* comes from the "global" view and the second read comes from the "texture" view which can make use of the texture cache as there are data reuses. The last one, which is a write to array *length*, goes to the "global" view. Without the multiview support, both the rule-based method and PORPLE can not do any optimization. As a result, Multiview achieves 1.13X speedup.

(3) Benchmark SP. SP is a program for survey propagation. The rule-based approach places all read-only data on texture memory and PORPLE does the same except for array *g_bias_list_vars*. Because of its regular accesses, it is put into global memory by PORPLE and suffers a lower latency. PORPLE outperforms the rule-based approach slightly. The main difference in the placement by Multiview is on the array *clauses.sat*. This array is both read and written in the kernel. As a result, both the rule-based method and PORPLE put it into the global memory. Similar to *BFS*, writes to this array are also idempotent. Our method recognizes the opportunity for coherence-free multiview and creates the "texture" view and "global" view for that array. Reads to that array go to the "texture" view, and writes go to the "global" view, leading to the best performance among all methods.

(4) Gaussian kernels. For the Gaussian benchmark, the rule-based method achieves 1.5X speedup for kernel *fan2*, but results in some slight slowdown to kernel *fan1*. Both PORPLE and Multiview outperform the rule-based method significantly. The reason is that the rule-based method tries to place all read-only arrays to the texture memory and leave all other arrays to the global memory. PORPLE, on the other hand, puts all arrays onto the global memory because of the access patterns. Moreover, for arrays whose accesses show a 2D data locality (e.g., *a_cuda, m_cuda*), PORPLE uses 2D surface API to access them, making the cache of the global memory support 2D spatial locality, giving 2.84X speedups.

Multiview, on the other hand, creates a "global" view and a "texture" view for array *a_cuda* such that the kernel *fan2* reads from 2D "texture" view and writes to the "global" view through the 2D surface API. The multiview is safe because all writes to that array happen after all reads to it. All reads to array *m_cuda* are from the 2D "texture" view, and all reads and writes to array *b_cuda* are from a "global" view. In this way, Multiview better utilizes the texture bandwidth and global bandwidth. It achieve 4.27X speedup, significantly larger than any other methods.

Through the NVIDIA profiling tool (nvvp), we observe that the *fan2* kernel is bandwidth bound. Table 3 shows the profiled bandwidth usage of the four different versions of gaussian_fan2. Since there are not many data reuses, the L2

Table 4: Placement decisions made by the rule-based approach, PORPLE and Multiview.
(T: texture memory, G: global memory, SF: surface memory; 1D: Array in one Dimension, 2D: cudaArray in two Dimensions)

	gaussian_fan1			gaussian_fan2					
References	1R-a_cuda	2R-a_cuda	1W-m_cuda	1R-a_cuda	2RW-a_cuda	1R-m_cuda	2R-m_cuda	1R-b_cuda	2RW-b_cuda
Rule-Based	T-2D	T-2D	G-1D	G-1D	G-1D	T-2D	T-2D	G-1D	G-1D
PORPLE-M2075	SF-2D	SF-2D	SF-2D	SF-2D	SF-2D	SF-2D	SF-2D	G-1D	G-1D
PORPLE-K20c	SF-2D	SF-2D	SF-2D	SF-2D	SF-2D	SF-2D	SF-2D	G-1D	G-1D
PORPLE-GTX980	SF-2D	SF-2D	SF-2D	SF-2D	SF-2D	SF-2D	SF-2D	G-1D	G-1D
Multiview-M2075	SF-2D	T-2D	SF-2D	SF-2D	SF-2D	T-2D	T-2D	T-1D	G-1D
Multiview-K20c	T-2D	T-2D	SF-2D	T-2D	SF-2D	T-2D	T-2D	G-1D	G-1D
Multiview-GTX980	T-2D	T-2D	SF-2D	T-2D	T-2D(r),SF-2D(w)	T-2D	T-2D	T-1D	G-1D

cache and device memory bandwidth affect the performance significantly. The original program has the lowest bandwidth usage. Multiview outperforms the other three significantly by utilizing all available bandwidth (texture and global) effectively, enjoying 188GB/s bandwidth of the texture cache, 271GB/s of the L2 cache and 126GB/s of the global memory.

(5) Lud kernels. The situations of the two lud kernels are similar to the *fan2* kernel in Gaussian. There is only one off-chip array m in kernel lud_perimiter and lud_internal. Both kernels write to m after reading it. Since m is not a read-only array, the rule-based method and PORPLE both place it in the global memory, offering no optimization to the kernels. Multiview creates a "texture" and a "global" view for the array; reads go to the former and writes go to the latter. (Shared memory has been used for some other arrays.) The better balance of global bandwidth usage and texture bandwidth usage helps improve the performance by 25% for lud_internal and 15% for lud_perimiter even though the latency of accessing the "texture" view is larger than that of accessing the "global" view.

(6) B+tree kernels. In this benchmark, there are two kernels named *findK* and *findRangeK*. Both the rule-based method and PORPLE place onto the shared memory the arrays *currKnodeD, keysD, lasstKnodeD, startD, endD*. The placement provides 1.19X and 1.23X speedup for the two kernels respectively. Multiview, on the other hand, notices that $knodesD\,[offsetD\,[bid]]\,.indices\,[thid]$ are used multiple times in the kernels and reads to it happen before writes to it. It hence creates a local view of them (by assigning them to local variables), which causes them to be stored in registers, speeduing up the two kernels by 1.35X and 1.32X.

Portability. Both PORPLE and Multiview are potentially able to customize the data placements to the memory systems of different GPUs, while the rule-based method cannot. For instance, Table 4 shows the placement decisions made by the three methods for kernels *gaussian_fan1* and *gaussian_fan2*. The row of "references" lists the reads and writes to the major arrays in the two kernels. For instance, "1R-a_cuda" means the first read reference to array a_cuda, and "2RW-a_cuda" means the second statement in the kernel accessing "a_cuda" and it conducts both read and write to it.

The placements on the three GPUs by the rule-based method are identical. PORPLE, although showing some cross-architecture adaptivity on some other kernels, happens to make the same decisions on the three GPUs for these kernels. The reference-discerning placement by Multiview allows more flexibility in tailoring placements to memory systems. It makes different placement decisions on the three GPUs for both kernels, helping them achieve the best performance.

Table 5: Speedups of Cross Runs.

		gaussian_fan1		gaussian_fan2	
		Placement for		Placement for	
		M2075	K20c	M2075	K20c
Run on	M2075	**1.28X**	1.19X	**2.23X**	3.61X
	K20c	1.14X	**1.27X**	1.92X	**4.27X**

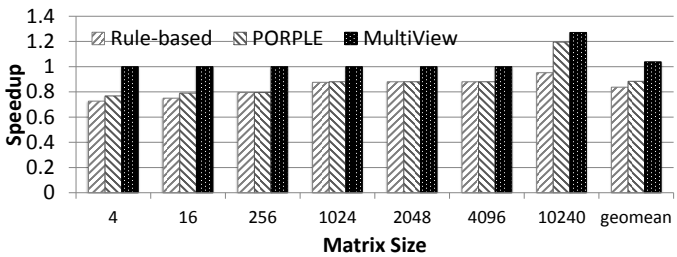

Figure 9: Speedup of Gaussian_fan1 on Different Inputs.

Using M2075 and K20c, Table 5 shows the performance results when running a kernel on a GPU with the data placement that Multiview finds on a different GPU. It clearly shows that the data placement it finds for a GPU does work better than what it finds on a different GPU. For example, running *fan1* with the placement generated for M2075 on K20c, it can only provides 1.19X speedup, significantly less than the 1.27X speedup achieved by using the placement found on K20c. The same holds on the *fan2* kernel. These results further confirm the capability of Multiview for automatically tailoring data placements to the memory system of a GPU.

Cross-Input Adaptivity. To study the input sensitivity, we test the *gaussian_fan1* and *gaussian_fan2* kernels on seven different inputs on K20c. Figures 9 and 10 show the results.

For *gaussian_fan1*, the number of thread blocks is the matrix size divided by 512. When matrix size is small(<10240), there are less than 20 thread blocks, which causes low occupancy, and the kernel is latency bound. PORPLE tries to minimize the total number of transactions by placing data in 2D locality surface memory. However, the 2D surface memory has a longer latency than the global memory (365clks vs. 330clks). That's why PORPLE even has a slight slowdown for *gaussian_fan1*. On the other hand, the rule-based approach tries to place all read-only data to 2D texture mem-

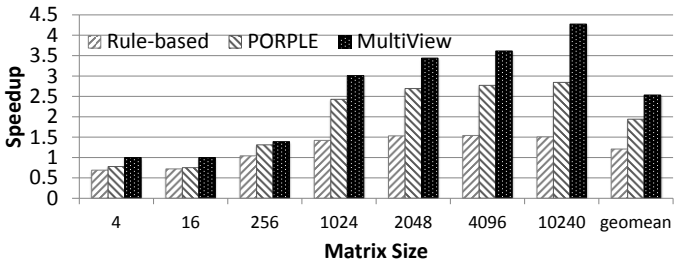

Figure 10: Speedup of Gaussian_fan2 on Different Inputs.

ory. Because the 2D texture access also has a longer latency than 1D global access (371ckls vs. 330clks), there is also some slowdown. The more flexibility in data placement in Multiview helps it strike a better balance in bandwidth and latency. When the total number of thread blocks is small ($\leq$ 8), it keeps all data in the global memory and uses only one view of each array. Once the input size reaches 10240, it uses the placements that have been shown in Table 4, achieving significant speedups. PORPLE is also able to adapt to input changes, but the speedups are limited by its single-view constraint as discussed before.

For *gaussian_fan2*, the number of thread blocks is $\left(\frac{Matrix\_Size}{16}\right)^2$. So, when input size is 4 or 16, the total number of thread blocks is less than 8, which makes the kernel latency bound. There are slowdowns for the rule-based method and PORPLE. The reasons are the same as explained for *gaussian_fan1*. As the input size increases, the rule-based approach, PORPLE and Multiview all achieve some speedups. The better flexibility in data placement by Multiview helps it outperform the other two methods significantly.

5.3 Other Applicability

We take the MVT benchmark from Polybench [8] as an example to demonstrate further applicabilities of the coherence-free multiview technique. In the benchmark, there are two kernels that are called one after the other without memory copy in between. The first kernel repeatedly updates array *x1* with the multiplication results of arrays *a* and *y_1*. The second kernel repeatedly updates array *x2* with the multiplication results of arrays *a* and *y_2*. A beneficial optimization to the program is to combine these two kernels into one and then fuse the two loops in the two kernels together, which reduces both the kernel launch and control-flow overhead, and also creates more opportunities for compiler optimizations. The fused kernel achieves 1.29X speedup compared to the original version.

On the fused kernel, the rule-based and PORPLE approaches both decide to place *y_1* and *y_2* on constant memory, and *a* on 2D texture memory. That placement provides 4.9X speedup compared to the original fused kernel. Multiview, on the other hand, places array *a* on 2D cudaArray, lets the first reference read from the view of 2D texture memory, and the second from the view of 2D surface memory. The two views allow a more effective usage of the bandwidth of the L1 and texture caches. With that enhancement, the kernel becomes 5.7X faster than the original fused kernel.

6. DISCUSSIONS

Slacks in coherence requirements in GPU programs are not limited to CUDA programs. In fact, OpenCL programs may expose even more opportunities. For instance, with the "scope" concept introduced in OpenCL 2.0, the atomic operations with a smaller scope reduces the strength of the synchronization requirement, and hence could potentially bring extra opportunities for applying the multiview data placements. Detailed explorations go beyond the scope of this paper.

Not every program has slack in coherence requirement. Our observations show that applications based on graphical structures (e.g., graphs, trees) often have certain slack in coherence for multiviews. In addition, some important scientific kernels tend to contain some reads-before-writes objects and other scenarios that are amenable for the multiview technique to apply. This paper shows that a set of important algorithms do show coherence slack and can substantially benefit from the proposed technique. As these algorithms are pivotal to a broad range of applications in graph, scientific computing and other domains, the impact of the technique can be significant. In addition, we show that code transformations (e.g., kernel fusion) can stimulate even more opportunities for coherence-free multiview.

As typical compiler analysis, the application of the static analysis in this work is conservative. If there are possible dependencies that prevent the multiview technique from being applied soundly to a data object, the compiler would not apply the transformations. Some more aggressive approaches could be worth future explorations.

7. RELATED WORK

The work closest to this study includes the rule-based data placement [3] and PORPLE [2], which have been compared with this work in earlier sections. Besides them, there are some work focusing on data placement in some particular settings or on a particular type of memory, rather than the general memory systems. For example, Wang and others have explored the energy and performance tradeoff in placing data on DRAM versus non-volatile memory [9]; Ma and Agrawal [10] propose the use of integer programming for the use of shared memory on GPU. Data placement can be important for CPU systems as well, especially for those that are equipped with heterogeneous memory architectures (e.g., 3D stacked memory [11], phase change memory [12, 13]). There has been some recent work on memory models for GPUs [14, 15, 16, 17]. They focus on race-free memory models for heterogeneous systems and GPU kernel race-free verifications.

To the best of our knowledge, this current study is the first work that supports multiviews of data objects on a complex GPU memory system; the reference-discerning data placement it enables overcomes limitations of prior methods and offers significant performance improvement.

Some studies have explored data layout for enhancing GPU program performance. For instance, Che and others have shown that conversions between row-major and column-major matrices can be beneficial on GPU [18]. Zhang and others [19] propose a runtime pipeline approach to improving array layouts for irregular memory accesses. There are some other works [20, 21, 22, 23, 24]. Data layout and data placement are orthogonal, one enhances data organiza-

tions on one type of memory, the other decides which type of memory to place which data. They could be used together.

Idempotent and other related properties have been used for other purposes, such as bug recovery [25], atomic-operations removal [7], avoidance of recovery in speculative executions or exception handling [26, 27, 28, 29]. We are not aware of prior usage of the properties for data placement.

8. CONCLUSION

This work introduces the concept of *coherence-free multiview*, and provides a simple theorem with which a compiler can safely apply *reference-discerning data placement*. By effective leveraging the slack in coherence requirement in GPU programs, the new technique significantly outperforms the state of the art, leading to 1.6X average (up to 4.27X) speedups. It further shows good portability across GPU models and program inputs. Given the continuously growing parallelism in processors, demands for memory performance keep increasing, which is expected to stimulate more sophisticated memory system designs. Coherence-free multiview open the new opportunities for software to tap into the full power of future memory systems.

Acknowledgment

We thank the anonymous reviewers for the helpful comments. This material is based upon work supported by DOE Early Career Award (DE-SC0013700), the National Science Foundation (NSF) under Grant No. 1455404, 1455733 (CAREER), and 1525609, and Google Faculty 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, NSF, or Google.

9. REFERENCES

[1] "NVIDIA CUDA." http://www.nvidia.com/cuda.

[2] G. Chen, B. Wu, D. Li, and X. Shen, "Porple: An extensible optimizer for portable data placement on gpu," in *Proceedings of the 47th International Conference on Microarchitecture*, 2014.

[3] 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.

[4] 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.

[5] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval, "Lonestar: A suite of parallel irregular programs," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[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] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on gpus," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, 2013.

[8] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *Innovative Parallel Computing (InPar), 2012*, pp. 1–10, IEEE, 2012.

[9] 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*, PACT '13, (Piscataway, NJ, USA), pp. 93–102, IEEE Press, 2013.

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

[11] 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*, pp. 404–415, 2013.

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

[13] 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*, ISCA '09, pp. 24–33, 2009.

[14] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson, "The design and implementation of a verification technique for gpu kernels," *ACM Trans. Program. Lang. Syst.*, vol. 37, pp. 10:1–10:49, May 2015.

[15] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free memory models," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, (New York, NY, USA), pp. 427–440, ACM, 2014.

[16] B. R. Gaster, D. Hower, and L. Howes, "Hrf-relaxed: Adapting hrf to the complexities of industrial heterogeneous memory models," *ACM Trans. Archit. Code Optim.*, vol. 12, pp. 7:1–7:26, Apr. 2015.

[17] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, "Remote-scope promotion: Clarified, rectified, and verified," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (New York, NY, USA), pp. 731–747, ACM, 2015.

[18] 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*, SC '11, pp. 13:1–13:11, 2011.

[19] 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.

[20] 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.

[21] 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*, pp. 225–234, 2008.

[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*, PACT '10, pp. 513–522, 2010.

[23] 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*, ICS '12, 2012.

[24] 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*, PLDI '10, pp. 86–97, 2010.

[25] W. Zhang, M. Kruijf, A. Li, S. Lu, and K. Sankaralingam, "Conair:featherweight concurrency bug reovery via single-threaded idempotent execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[26] M. de Kruijf and K. Sankaralingam in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[27] S. A. Mahlke, W. Y. Chen, W. Hwu, B. Rau, and M. Schlanskar, "Sentinel scheduling for vliw and superscalar processors," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[28] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. Vijaykumar, "Exploiting reference idempotency to reduce speculative storage overflow," *ACM Transactions on Programming Languages and Systems*, vol. 28, pp. 942–965, September 2006.

[29] M. Hampton, *Reducing Exception Management Overhead with Software Restart Markers*. PhD thesis, MIT, 2008.