Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-Coalesced Memory Accesses on GPU [*]

Bo Wu[⋆], Zhijia Zhao[⋆], Eddy Z. Zhang[†], Yunlian Jiang[‡], Xipeng Shen[⋆]
[⋆]The College of William and Mary, Williamsburg, VA, USA
[†]Rutgers University, NJ, USA
[‡]Google, USA
{bwu,zzhao}@cs.wm.edu, eddy.zhengzhang@cs.rutgers.edu, yunlian@google.com, xshen@cs.wm.edu

Abstract

The performance of Graphic Processing Units (GPU) is sensitive to irregular memory references. Some recent work shows the promise of data reorganization for eliminating non-coalesced memory accesses that are caused by irregular references. However, all previous studies have employed simple, heuristic methods to determine the new data layouts to create. As a result, they either do not provide any performance guarantee or are effective to only some limited scenarios. This paper contributes a fundamental study to the problem. It systematically analyzes the inherent complexity of the problem in various settings, and for the first time, proves that the problem is NP-complete. It then points out the limitations of existing techniques and reveals that in practice, the essence for designing an appropriate data reorganization algorithm can be reduced to a tradeoff among space, time, and complexity. Based on that insight, it develops two new data reorganization algorithms to overcome the limitations of previous methods. Experiments show that an assembly composed of the new algorithms and a previous algorithm can circumvent the inherent complexity in finding optimal data layouts, making it feasible to minimize non-coalesced memory accesses for a variety of irregular applications and settings that are beyond the reach of existing techniques.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—optimization, compilers

General Terms Performance, Experimentation

Keywords GPGPU, Memory coalescing, Computational complexity, Thread-data remapping, Runtime optimizations, Data transformation

1. Introduction

Recent years have seen a rapid adoption of Graphic Processing Units (GPU) for high performance computing. As a massively parallel architecture, GPU significantly accelerates many regular, data-parallel applications. But its benefits for irregular applications are far less substantial, especially when the application contains dynamic, irregular memory references.

```
(a) codelet
        // tid: the global ID of a thread
        // M: num. of neighbors per molecule
        ipos = pos [tid];
        for (j=0; j< m; j++){
            jpos = pos [ neighbors [ j*M + tid]];
            computeForce (f, ipos, jpos);
        }
        force [tid] = f;
(b) case 1: neighbors [0...3] = {4, 5, 6, 7}
(c) case 2: neighbors [0...3] = {9, 103, 23, 67}
```

Figure 1. (a) A simplified codelet of the force computation in a molecular dynamics simulation. The values in *neighbors* decides the access pattern of *pos*. (b) and (c) show a regular and irregular pattern respectively.

The reason comes from the hardware properties of GPU. GPU organizes its threads in groups and memory in segments. Every W threads with consecutive ID numbers form a *warp*; every S consecutive bytes in the global memory form a *segment*. At a memory reference, the number of memory transactions needed to load the data accessed by a warp equals the number of segments the data fall onto. When that number is larger than the possible minimum, the accesses are called *non-coalesced memory accesses*.

Non-coalesced memory accesses are common in irregular applications. Figure 1 (a) shows a simplified codelet in the core computation in a molecular dynamics simulation. The underlined statement "pos [neighbors [j*M + tid]]" gets the coordinates of a neighbor of the current molecule. As a typical dynamic irregular reference, it may manifest various access patterns, determined by the values contained in *neighbors*. In the case of Figure 1 (b), all accesses by the warp are to a single memory segment; only one memory transaction is necessary, assuming a segment can contain four molecules' positions. But in the case of Figure 1 (c), because of irregular values in *neighbors*, the accesses are non-coalesced and require four memory transactions. This kind of irregularity is common in a molecular dynamics simulation, thanks to molecules' movements and their dynamic neighborhood. It is a key feature of many scientific simulations.

[*] Zhang and Jiang worked on this project before their graduations from The College of William and Mary.

PPoPP'13, February 23–27, 2013, Shenzhen, China.
Copyright © 2013 ACM 978-1-4503-1922-5/13/02...$15.00.

Non-coalesced accesses may result in memory transactions as many as W times of the minimum, leading to a throughput a number of factors lower than the peak of GPU [2, 5, 26]. They have been the focused target of some recent studies. However, most prior explorations [2, 12, 18, 20, 25] concentrate on static irregularities, where the memory access patterns are known at compilation time. The type of irregularity in our focus is dynamic: For instance, the content of the indexing array *neighbors* in Figure 1 depends on the input to the program and is updated throughout the simulation of the molecules movement.

Dynamic irregular accesses have to be treated during runtime. Some earlier studies [22] have relied on special hardware extensions that modern GPUs do not have. A recent study [26] shows the promise of pure software solutions. It develops a pipeline scheme that makes it possible for CPU to reorganize data and threads for a near-future GPU kernel invocation while GPU is working on the current kernel. A related study [5] demonstrates the feasibility of moving the reorganization to GPU so that CPU can also involve in workload processing.

Despite that these studies have shown promising speedups, the understanding to data reorganization for minimizing non-coalesced GPU memory accesses remains preliminary. The focus of the previous studies has been on coordinating CPU and GPU to allow runtime data reorganization. They have not explored fundamental issues on data reorganization and its relation with GPU memory performance. As a result, the reorganization algorithms they have adopted either lack performance guarantees or are effective to only some limited scenarios, as Section 3 reveals.

This current work provides the first principled understanding of GPU data reorganization for minimizing non-coalesced accesses. It makes four-fold contributions.

- **Complexity Analysis:** It systematically analyzes the relations between data reorganization and GPU memory accesses, and proves it infeasible to minimize memory transactions in polynomial time through just data repositioning, unless P equals NP. Furthermore, it proves that even if threads are allowed to be regrouped into warps, the complexity remains unchanged. The results indicate that it is virtually in vain to keep searching for a general, practical algorithm that can minimize GPU memory accesses, either with or without hardware extensions, through data repositioning, thread regrouping, or their combination—the three methods that have been pursued by most previous GPU memory optimizations. The strong theoretical results are essential for guiding the directions of the current research efforts. (Section 2)
- **Limitations and Essence:** This work points out the limitations of existing algorithms for optimizing GPU dynamic irregular references, and unveils that in practice, the essence for designing an appropriate data reorganization algorithm can be reduced to a classical tradeoff among space, time, and complexity. (Section 3.1 and 3.2)
- **Algorithms:** Based on the insights, this work develops two new data reorganization algorithms that complement prior algorithms with respective strengths. It shows that the new algorithms reduce space cost significantly with non-coalesced memory accesses kept minimized. (Section 3.3 and 3.4)
- **Comparison and Selection:** This work compares the various algorithms, unifies them into an assembly, and develops some selection guidelines and an automatic algorithm selector to address GPU dynamic irregular accesses in various scenarios. (Section 4)
- **Evaluation:** Experiments on a set of dynamic irregular applications show that the developed assembly, along with the algorithm selector, circumvents the inherent complexity in finding optimal data layouts, making it feasible to minimize non-coalesced memory accesses for a variety of irregular applications and settings that are beyond the reach of existing techniques. Compared to existing techniques, the assembly produces up to 2X speedup (10-50% on average), demonstrating its promise as a comprehensive solution to dynamic irregular memory accesses in GPU. (Section 5)

2. Problem Setting and Complexity Analysis

In this section, we first provide some background on GPU that closely relates with the following discussions. We then describe the main approaches researchers have been pursuing to tackle non-coalesced GPU accesses. We finally reveal the fundamental challenges for such approaches by proving that using those approaches to minimize non-coalesced accesses for general irregular references is computational infeasible unless NP equals P.

2.1 Background on GPU

As a massively parallel device, GPU contains hundreds of cores residing on a number of streaming multiprocessors (SM). When a GPU kernel is launched, the runtime usually creates thousands of GPU threads running on these cores in parallel. These threads are organized hierarchically. A number of threads (32 in NVIDIA GPU) with consecutive IDs compose *a warp*, a number of warps compose *a thread block*, and all thread blocks compose *a grid*. (This paper uses CUDA terminology.) A warp is the unit in GPU scheduling; all threads in a warp proceed in lockstep.

GPU is equipped with several types of memory. The largest is off-chip main memory called *global memory*. It consists of a large number of segments (of 32, 64, or 128 bytes depending on the access mode.) For the large size and long access latency of global memory, its access efficiency is critical. GPU hence offers *memory coalescing*, a hardware-enabled feature that uses one memory transaction to load/store all the data in a memory segment that are requested by a warp at a load/store instruction. As a result, the execution of a load/store instruction by a warp incurs K memory transactions, where K equals the number of memory segments the requested data fall onto. Suppose the data to load/store by a warp at a reference contain D bytes and a memory segment is S-byte long. The reference is a *non-coalesced reference* when $K > \lceil D/S \rceil$. The corresponding memory accesses are *non-coalesced accesses*. Another type of memory on GPU worth mentioning is *shared memory*, which is on-chip with access latency comparable to that of register files. A thread can access an element that is loaded or stored into shared memory by another thread if and only if the two threads belong to the same thread block. In the following discussion, memory refers to global memory by default.

2.2 Objective and Complexity

The objective of non-coalesced access minimization is to minimize the number of non-coalesced accesses of a GPU kernel. The minimization, for its importance for GPU performance, has drawn lots of attentions. However, satisfying solutions are still limited to some special scenarios. In this section, we examine the inherent complexity of the previous approach and prove that in general cases, using the approach is infeasible to reach the objective unless NP equals P. The results may guide the direction of future research, and also lays the theoretical foundation for the rest of this work.

A GPU kernel may contain multiple references. We focus on one reference first and discuss other scenarios later.

2.2.1 Data Repositioning and NP-Completeness

Data repositioning has been the main direction pursued by previous work for minimizing non-coalesced accesses [2, 18, 22]. The essential idea is to reorder data elements on memory so that the data to be accessed by a warp can reside consecutively, covering the minimum number of memory segments. For the example in Figure 1 (c), the transformation would create a new array *Pos'* with the same elements as *Pos* has but in a different order, such that the four elements *Pos[9], Pos[103], Pos[23], Pos[67]* fall into a single memory segment. Matrix transposing [22] is another example: By repositioning elements on memory to create a column-major data layout, it can minimize non-coalesced accesses for a column-wise reference to the matrix.

Although it seems simple, using data repositioning can be complicated when the data accessed by multiple warps overlap. Consider a reference $A[P[tid]]$, with P as follows
P[]={**8**, 23, 46, 93, **8**, **9**, 10, **67**, 5, 11, **41**, **67**, **9**, **41**, 55, 59}.
Assume memory segment length $S = 4$ and warp size $W = 4$. The repetitive values in P (highlighted in bold font) dictate that some elements in A are accessed by multiple warps. Which segment to put those values is tricky. For instance, putting A[8] into a segment with A[23], A[46], A[93] would coalesce the first warp's accesses but leave the second warp's accesses non-coalesced.

The issue has been largely limiting the applicability of data repositioning. Despite many recent efforts, this approach has been effective for only the cases where each target data element is accessed by only one warp in a kernel. In an application with dynamic irregular references, that condition rarely holds: In a molecular dynamics, a molecule is often the neighbor of more than one molecule; in a sparse matrix multiplication, an element in the vector is often used to multiply multiple elements in the matrix; in a mesh simulation, a vertex is often shared by several triangles.

Complexity Theorem Prompted by the various difficulties people have so far encountered in finding a general data repositioning algorithm to guarantee minimum non-coalesced accesses, we conduct a systematic analysis on the inherent complexity of the problem. An important finding we obtain is that such an algorithm does not exist unless NP=P. Formally, we develop the following theorem:

Theorem 1. *Creating a new data layout through only data repositioning (which implies that each item in the original data structure has only one copy in the new structure) to minimize the non-coalesced accesses for an arbitrary data reference on GPU is an NP-complete problem.*

As this is the first strong claim on the complexity of non-coalesced access minimization, it is worth providing a formal proof, for verifying its correctness as well as offering insights that may be useful for analyzing the complexity of other GPU optimization problems.

Proof Assume that the irregular reference, when executed by all the GPU threads, accesses n unique data items. Let z be the length of a memory segment (in the unit of data items.) The goal of the repositioning is essentially to partition the n data items evenly into n/z clusters such that when each cluster is put onto a single memory segment, the total number of memory transactions at that data reference is minimized. To prove the NP-completeness, we use the following notations.

Δ : the set of all data items to be partitioned; $n = |\Delta|$; Ψ : the set of all warps; $m = |\Psi|$; $\Psi_{<x>} = \{w | w \in \Psi$ & w references $x\}$ ($x \in \Delta$); Ω : a complete partition of Δ with z data items per cluster; $r_C = |\bigcup_{x \in C} \Psi_{<x>}|$ ($C \subseteq \Delta, |C| = z$).

We can see that $\bigcup_{x \in C} \Psi_{<x>}$ essentially contains all and only the warps that each accesses at least one element in C. So, essentially, r_C is the number of memory transactions incurred by all data items in C when they are put into a single memory segment. Hence, totally there are $\sum_{C \in \Omega} r_C$ memory transactions for partition Ω if each cluster of data is put into one memory segment. The target problem is to find an Ω such that $\sum_{C \in \Omega} r_C$ is minimized. Its corresponding decision problem is: Given an arbitrary number u, whether an Ω exists such that $\sum_{C \in \Omega} r_C \leq u$. We call this decision problem **DLDP** (Data Layout Decision Problem).

Reduction from 3DM (three-dimensional matching), a known NP-complete problem [11], to DLDP is enough to prove that our target problem is NP-hard. **3DM** is defined as follows.

- Input: 3 disjoint sets R, G, B, $|R| = |G| = |B| = l$, and a set of 3-D vectors T, $T \subseteq \{< r, g, b > | r \in R, g \in G, b \in B\}$.
- Problem: Is there a set S meeting all these conditions (3DM conditions): (1) $S \subseteq T$; (2) $|S| = l$; (3) ∀ ⟨r,g,b⟩∈ S, ∀ ⟨r',g',b'⟩∈ (S - ⟨r,g,b⟩), $r \neq r', g \neq g', b \neq b'$.

From an instance of 3DM problem, we construct an instance of DLDP as follows: $\Delta = |B \cup G \cup P|$, $n = 3l$; $m = |T|$; $z = 3$; $u = l(m-1)$; Ψ : a set of m warps, with each warp having a unique ID equaling one element in T, and a warp with ID ⟨r,g,b⟩ accesses x if and only if $x \in \Delta$, $x \neq r$, $x \neq g$, and $x \neq b$.

We prove that a solution, represented as $\hat{\Omega}$, to the constructed DLDP solves the 3DM problem. Because $\hat{\Omega}$ is a partition of Δ, $|\hat{\Omega}| = l$; because $\hat{\Omega}$ solves the 3DM problem, $\sum_{C \in \hat{\Omega}} r_C \leq u$. From $\hat{\Omega}$ we derive a set of 3-D vectors $\hat{S}$ ($|\hat{S}| = |\hat{\Omega}|$). Each element in $\hat{S}$, $< y_1, y_2, y_3 >$, comes from one element in $\hat{\Omega}$, $\{x_1, x_2, x_3\}$, with "y_1, y_2, y_3" as an ascending sequence of "x_1, x_2, x_3" in R,G,B order (i.e., $\forall r \in R, g \in G, b \in B, r < g < b$).

We prove that $\hat{S}$ is a solution to the 3DM problem—that is, it meets all the 3DM conditions. It is obvious that $\hat{S}$ meets condition two and three given its derivation from $\hat{\Omega}$. To prove the first condition, we need to show $\forall \vec{v} \in \hat{S} \Rightarrow \vec{v} \in T$. This step uses the following lemma: $\vec{v} \in T \Rightarrow r_{\vec{v}} = m - 1$; $\vec{v} \notin T \Rightarrow r_{\vec{v}} = m$.

The correctness of the lemma is easy to see if one notices that $\vec{v} \notin T$ means all warps in Ψ must access at least one element in $\vec{v}$ (hence $\bigcup_{x \in \vec{v}} \Psi_{<x>} = \Psi$, $r_{\vec{v}} = m$), while $\vec{v} \in T$ means all but one warp whose ID equals $\vec{v}$ access at least one element in $\vec{v}$.

From the lemma, we see that if $\exists \vec{v}\ \vec{v} \in \hat{S}$ and $\vec{v} \notin T$, then $\sum_{C \in \hat{\Omega}} r_C$ must be greater than u, $u = l(m-1)$, contradicting the initial condition that Ω is a solution to the DLDP. Thus, DLDP is NP-hard. Apparently, DLDP belongs to NP; the optimal data layout problem is hence NP-complete. Theorem 1 hence follows.

2.2.2 When Warp Reorganization is Allowed

The above proof assumes that only data repositioning is applied for reducing non-coalesced memory accesses. Some recent study [26] has shown that warp reorganization can help remove non-coalesced accesses as well, and can be used together with data repositioning for the optimization. In this subsection, we complement Theorem 1 by proving that using warp reorganization does not change the NP-completeness of the problem. That is, we prove the following strengthened complexity theorem:

Theorem 2. *It is an NP-complete problem to minimize the non-coalesced accesses for an arbitrary data reference on GPU through data repositioning, warp reorganization, or both.*

Warp reorganization is to swap threads across warps. It is also called *job swapping* because it exchanges the jobs of swapped threads and hence the data elements a warp accesses. The swapping may remove non-coalesced accesses. For instance, suppose that thread t_3 accesses $A[7]$ and thread t_7 accesses $A[3]$, while the other threads in the first two warps access $A[tid]$. After swapping t_3 and t_7, the new t_3 will do the work originally done by t_7 and access $A[3]$, while the new t_7 will take over the work of t_3 and access $A[7]$. Both warps' accesses become coalesced. Runtime warp reorganization has been shown to be feasible through either hardware extensions [9] or program transformations [26]. It may be used together with data repositioning in minimizing non-coalesced accesses [26].

We now prove that allowing warp reorganization does not change the computational complexity of the data repositioning problem. To prove it, it is enough to prove a special case of the problem to be NP-complete. The special case we use is when each data element is of the same size as the memory segment. In that case, data repositioning has no effect on the number of memory accesses as it does not change the clustering of data elements into memory segments. So if we can prove that using only warp reorganization to minimize non-coalesced memory accesses is NP-complete, Theorem 2 is proved.

Warp reorganization is equivalent to grouping the jobs of the threads into clusters with each cluster containing W jobs (W is the number of threads per warp). Without loss of generality, consider that a job contains just one irregular reference to an array. Let N stand for the number of threads, M represents the total number of memory segments that contain at least one data item requested at the reference. We claim that it is NP-complete (in regard to M) to partition N jobs evenly into N/W clusters such that when each warp takes one cluster of jobs, the total number of memory transactions at that data reference is minimized.

The proof is via a reduction from a known NP-complete problem, the **partition problem** [11] (represented as **PAR** to avoid confusion). All jobs can be classified into M categories based on which memory segment contains the data item requested in a job; we say two jobs are of the same type if they are in the same category. Let n_i ($i = 1, \cdots, M$) represent the number of jobs in the i^{th} category. Apparently the total number of jobs N equals $\sum_{i=1}^{M} n_i$. In the PAR problem, the goal is to decide whether a given set S of integers can be partitioned into two subsets S_1 and S_2 such that the sum of the numbers in S_1 equals the sum of the numbers in S_2. The reduction to our problem is as follows. A special case of our problem is that the size of each warp (W) is $N/2$. Our remapping hence needs to assign the N jobs to two clusters. Apparently, the number of memory transactions for a job cluster is the number of types of jobs in the warp. It can be seen that the achievable lower bound of the number of memory transactions in our special-case problem is M. It is obtainable only when $M/2$ types of jobs fit exactly into one cluster; note that in this case, the integer set $S = \{n_1, n_2, \cdots, n_M\}$ is evenly partitioned. So, if we can find the best partition of threads into warps in polynomial time (in regard to M), we would be able to tell whether S can be evenly partitioned by checking whether the number of memory transactions resulting from our mapping is M. Hence, the PAR problem would be solvable in polynomial time, contradicting the well-established NP-completeness of the problem. Theorem 2 follows.

2.3 Discussion

This section has analyzed the computational complexity of using data repositioning for minimizing non-coalesced accesses. The proved NP-completeness should not rule out the possibility that through some heuristic algorithms, the approach may still yield good speedup on some special types of kernels. However, it does indicate the extreme challenge to use it for achieving the optimal for general cases. We next show that the challenge can be circumvented if a constraint assumed by the approach is relaxed.

3. Algorithms that Circumvent the Complexity

We design two new algorithms to circumvent the complexity facing data repositioning. The key observation is that the essential difficulty in data repositioning comes from an implicit constraint that the produced new data layout uses no more space than the original. If we allow more space to be used, the complexity of the problem may reduce significantly.

Previous studies have not exploited this insight, except for the one by Zhang and others [26], which takes advantage of extra space but in an ad-hoc manner. In this section, we first review that previous method, reveal its limitation, and crystallize the analysis into an insight in the key tradeoff in designing a practical solution. We then describe the two new algorithms we design.

The following discussion is based on reference $A[P[tid]]$, a conceptual form of dynamic irregular references. It assumes the memory segment size (S) is a multiple of the working set size of a warp. This condition often holds given that the warp size and S are typically powers of 2. But even when it does not hold, the following discussions are still valid except that some preprocessing needs to be done to align data with memory segments.

3.1 Review of the *Duplication* Algorithm

The *duplication algorithm* is used by Zhang and others to optimize irregular memory accesses [26]. For an irregular reference, such as $A[P[tid]]$, the algorithm creates a new array A' such that $A'[tid] = A[P[tid]]$; the reference to $A[P[tid]]$ in the kernel is then replaced with $A'[tid]$. The algorithm naturally ensures that all accesses of a warp are to a consecutive memory region and there are no non-coalesced memory accesses, as illustrated by Figure 4 (b).

The algorithm is called "duplication" as it creates duplicated copies of a data element when the indexing array P contains repetitive values. Apparently, the new array A' is as large as the number of GPU threads (T), no matter how small the original array A is. Even worse is when there are multiple references to the same array (e.g., $A[P[tid]] + A[P[tid] + v]$), the algorithm creates a new T-long array for each of the references.

3.2 Limitations and Tradeoff

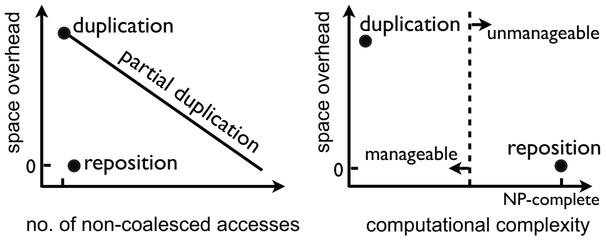
The duplication algorithm converts irregular accesses to regular ones. However, it may dramatically inflate space usage. For a K-element array referenced n times by T GPU threads, the space overhead is as much as a factor of $n * T/K$. In a modern GPU, T can be comparable with the number of bytes in the entire memory.

The large space overhead has two consequences. First, the basic duplication algorithm fails to apply when the space inflation exceeds the capacity of the memory. Second, the creation and transfer of the large volume of data may introduce substantial time overhead, throttling the optimization benefits. The previous work has used partial duplication to alleviate the issues [26]. The idea is to apply the transformation to only a fraction of the GPU threads. Although it can reduce the space overhead, it compromises the quality of the optimization proportionally. As Section 5 will show, it may result in substantially lower speedup than what is possible.

Figure 2 shows the conceptual positions of the previous approaches in a space of optimization quality, complexity, and space cost. Data repositioning and the duplication algorithm are at two extreme ends of the spectrum of space cost. The partial duplication lowers the space cost but also proportionally degrades the transformation quality and lengthens the kernel execution time. Data repositioning has the lowest space cost but the highest complexity. So the key for having a practical algorithm is to find a sweet design point that reduces the space cost without compromising the transformation quality and meanwhile possesses manageable complexity. We next describe two new algorithms towards that goal.

3.3 *Padding* Algorithm

The padding algorithm tries to avoid some unnecessary data copies made in the duplication algorithm without compromising the optimization quality. Its basic observation is that if two threads (t_1 and t_2) from the same warp access the same data element (a), there is



(a) space-quality space (b) space-complexity space

Figure 2. Positions of various algorithms in the space-quality-complexity coordinates. Graph (b) omits partial duplication for legibility.

```
// inputIndArray, outputIndArray: the original and produced indexing arrays
// inputArray, outputArray: the original data array and its new copy after padding
function Padding(inputIndArray, outputIndArray, inputArray, outputArray)
 inputIndArray = SortByFrequency(inputIndArray);
 for each warp w,
   uniqueRefSet = FindUniqueRefs(w, inputIndArray, inputArray);
   nRemainingSlots = FindRemainingSlots(currentMemSegment);
   if uniqueRefSet.size <= nRemainingSlots,
     for each e in uniqueRefSet,
       add element e to outputArray;
       update outputIndArray;
     end
   else
     pad dummy values to outputArray for memory segment alignment;
     for each e in uniqueRefSet,
       add element e to outputArray;
       update outputIndArray;
     end
   end
 end
end
```

Figure 3. The pseudo-code of the padding algorithm.

no need to create two copies of that data element. We can simply let them access the same copy of the data element. It will change the one to one regular mapping between data and threads created by the duplication algorithm, however, it will not create non-coalesced accesses since the two threads still access the same memory segment.

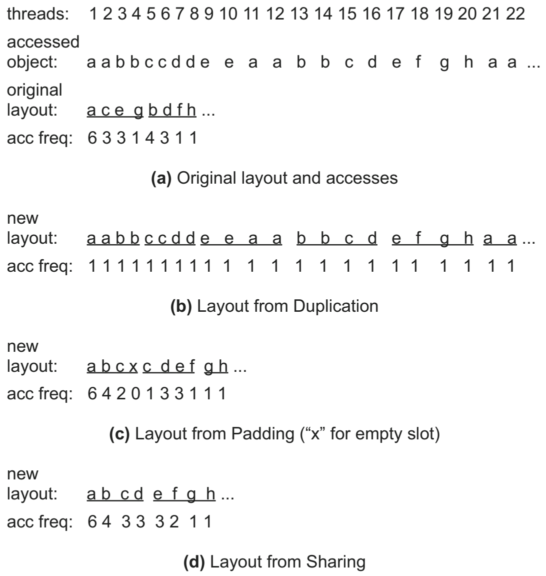
3.3.1 A Simple Design

A simple design is to just make the following modification to the duplication algorithm. When the algorithm is about to create a copy of an element in the new array A', it checks whether the current thread is the first of the current warp that accesses that element and avoids the creation if it is not. (An entailed change is that the original reference, say $A[P[tid]]$, needs to be replaced with $A'[Q[tid]]$ rather than $A'[tid]$, where Q contains the new mapping between a thread and the data it accesses in A'.)

Unfortunately, this simple modification is insufficient for two reasons. First, the avoided duplications cover only a small portion of all the duplications because the chance for two threads accessing the same data element to come from the same warp is often small. Second, the avoidance of some duplications often causes a misalignment between the working set of a warp and memory segments. As a consequence, the working set of a warp may span over the boundary of a segment, causing new non-coalesced accesses.

3.3.2 An Enhanced Design

Our enhanced design addresses the two problems of the simple design through sorting and padding. Figure 3 shows the pseudo code of the algorithm. It includes three steps.


threads: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
accessed object: a a b b c c d d e e a a b b c d e f g h a a ...
original layout: a c e g b d f h ...
acc freq: 6 3 3 1 4 3 1 1
new layout: a a b b c c d d e e a a b b c d e f g h a a ...
acc freq: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
new layout: a b c x c d e f g h ...
acc freq: 6 4 2 0 1 3 3 1 1 1
new layout: a b c d e f g h ...
acc freq: 6 4 3 3 3 2 1 1

(a) Original layout and accesses

(b) Layout from Duplication

(c) Layout from Padding ("x" for empty slot)

(d) Layout from Sharing

Figure 4. An example that illustrates the algorithms of duplication, padding, and sharing. Assume 4 objects per memory segment, 4 threads per warp, and 4 warps per block.

The first step reorders data elements based on their access frequencies. At a data reference, the *access frequency* of a data element is the number of threads that access it. The second step reorganizes threads into warps. It reorders threads according to the order of data elements—that is, all threads accessing data X must precede all threads accessing Y if X precedes Y in the new data sequence. Starting from the first thread, every W adjacent threads form a warp in the resulting thread organization. These two steps address the first problem of the simple design: By making threads accessing the same data element locate closely and form warps, they reveal more opportunities for saving duplications.

The third step puts data elements into memory segments. Starting from the first data element in the new order, the data are greedily packed into a memory segment one after one. But when it finds that the current segment cannot hold all the data the current thread warp accesses (e.g., the first segment in Figure 4 (c)), it moves all the data accessed by that warp into the next memory segment, leaving some empty slots at the end of the previous memory segment. Data is duplicated only when necessary—that is, when one data element is accessed by multiple thread warps whose working data sets do not fall into the same memory segment. Examples are the two "c"s in the layout in Figure 4 (c). This step addresses the second problem of the simple design. By padding a memory segment with some empty slots when necessary, it aligns the working set of a warp with memory segments.

Analysis This padding algorithm guarantees zero non-coalesced access since it puts the working set of every warp into a single memory segment. Its space overhead comes from the padded empty slots and some duplicated data elements. If k threads in a warp access one single data element, the empty slot in a memory segment is at most as long as $mod(S, \lceil W/k \rceil)$, where $\lceil W/k \rceil$ computes the number of unique elements accessed by a warp and S is the number of data elements a segment can contain. Both S and W are usually power of two. So the worst case happens when k is small (hence the remainder is large) but is not a power of two. Specifically, when $k = 3$, the empty slot is the longest, up to $W/3 - 1$. But even in that case, the space cost is much lower than that of the duplication algorithm. The number of threads a memory segment serves in that case must be no fewer than $t = (S - W/3 + 1)/(W/3)$. Following the assumption that S is the multiple of W, let $S = r * W$ with r being a positive integer. The number of threads served, t, must be no smaller than $3r - 1$, which is at least 2. In the duplication algorithm, these threads would use at least $2S$ memory (given that $r = 1$ means $S = W$), double what they use in the padding algorithm.

When analyzing the number of duplications in the padding algorithm, it is important to notice that among all the warps accessing the current memory segment, only the first of them may have some data elements duplicated. It is because only when the working set of a warp overlaps with the data elements in the previous memory segment, those overlapped elements may have a duplicate in the current memory segment (e.g., the second "c" in Figure 4 (c).) That overlap must be partial since at a complete overlap, the previous memory segment can already hold the working set of that first warp, and hence that warp would have corresponded to the previous rather than current memory segment. Due to the way threads are ordered, that partial overlap entails that the working sets of the other warps cannot overlap with the data in the previous segment, and hence have no duplicated data. Following the observation, we can see that in the case mentioned in the previous paragraph, the duplicated part of a segment is at most $W/3 - 1$, smaller than $1/3r$ of S. In comparison, the duplication algorithm creates at least 3 copies per data element in that case. The data element contained in one memory segment in the padding result would become $3 * (S - W/3 + 1)/S$ (which is greater than $3 - 1/3r$ and 2.67) segments in the result from the duplication algorithm.

Limitation Despite its appealing properties, the padding algorithm has one major limitation. Because it reorganizes not only data but also threads, it may cause side effects to other references in the kernel. For example, if a kernel contains "B[tid]+A[P[tid]]", after the third and ninth threads switch positions, they swap their jobs, and the accesses to B must also be swapped. In another word, $B[tid]$ must be replaced with $B[R[tid]]$ where, $R[3] = 9$, and $R[9] = 3$. Otherwise, the new thread 3 would add $A[P[9]]$ with $B[3]$ rather than $B[9]$, causing wrong computation results. As a result, the optimization of $A[P[tid]]$ makes accesses to B non-coalesced. So the padding algorithm is most beneficial when all references in a kernel follow the same access pattern (e.g., $B[P[tid]] + A[P[tid]]$.)

3.4 *Sharing* Algorithm

This algorithm overcomes the limitation of the padding algorithm by increasing duplication avoidance from a different angle. It uses the shared memory in GPU to enlarge the scope of duplication avoidance. Shared memory is a type of on-chip memory in GPU. Data written to shared memory by a thread is visible only to threads in the same thread block. Shared memory has an access latency a hundred times smaller than that of the global memory, and more importantly, its performance is largely insensitive to irregularities in accesses.

Insight The key insight of this algorithm is to shift irregular accesses from global memory to shared memory. As shared memory is visible within a whole thread block, the sharing algorithm enlarges the scope of duplication avoidance from a warp to a thread block. Its basic idea is to create a copy of all the data accessed by a thread block (a single copy per data element) and put them into a consecutive chunk of memory. Then, it loads these data in a consecutive (hence coalesced) manner into shared memory. It redirects memory accesses by the thread block to the corresponding copies in the shared memory. By keeping only one copy for all data elements accessed by a whole thread block, it avoids many duplications. By shifting irregular accesses to shared memory, it eliminates non-coalesced accesses to global memory. It uses clustering to further increase the opportunity for duplication saving. The detailed algorithm is as follows.

```
// inputIndArray, outputIndArray: the original and produced indexing arrays;
// inputArray, outputArray: the original and produced data arrays;
// blockPos, blockSizes: the starting position and size of the working set of a thread block
function Sharing(inputIndArray, outputIndArray, inputArray, outputArray,
blockPos, blockSizes)
  [inputIndArray, inputArray] = DataClustering(inputIndArray, inputArray);
  for each thread block b,
    uniqueRefSet = findUniqueReferences(b, inputIndArray, inputArray);
    blockSizes[b] = uniqueRefSet.size;
    blockPos[b] = outputArray.size;
    for each e in uniqueRefSet,
      Add element e to outputArray;
      outputIndArray[e] = position(outputArray, e) - blockPos[b];
    end
    pad dummy values to outputArray for memory segment alignment;
  end
end
```

Figure 5. The pseudo-code of the sharing algorithm.

Algorithm Figure 5 outlines the pseudo-code of the sharing algorithm. It includes two steps. In the first step, it conducts clustering to swap threads among thread blocks so that the threads in a block have as many accesses to the same data elements as possible, while different blocks have as few as possible.

Many clustering algorithms can serve for the purpose. In our implementation, we use two. The first is a graph partition-based clustering [23], which is especially suitable for applications with a graph topology, such as the distribution of molecules in a molecular dynamics simulation, the structure of a mesh in a mesh simulation. In these applications, typically each thread is in charge of one node in the graph. The algorithm randomly selects some nodes as seeds and assign each of them a distinct cluster number. The nodes then iteratively propagate the cluster memberships to their neighbors. The threads are clustered by inheriting the cluster ID of their corresponding nodes. The second clustering algorithm is suitable for other cases. It uses the working set of a thread as its feature and applies the standard hierarchical clustering to build up the clusters.

After clustering, the second step prepares data to be loaded into shared memory and creates a new indexing array to reference them. Specifically, it places the data elements accessed by each thread block continuously into a global array. It is possible that even after the clustering, the working sets of two thread blocks may still overlap. In that case, some data will have to be duplicated across thread blocks. Some trivial padding fills up the final memory segment a thread block uses. Figure 4 (d) shows an example. The clustering switches threads 9 and 10 with threads 21 and 22. After that, each thread block accesses four unique data elements and there is no overlaps between the working sets of the two blocks and hence no duplications. Two meta-arrays, *blockPos* and *blockSizes*, record the starting offset and the number of accessed data elements in the new data array for each thread block. They add minor space cost. When the GPU kernel executes, each thread block loads the corresponding block of data to shared memory according to the meta-arrays.

Notes We make several notes. First, the clustering step is optional. It increases the chance for saving data duplications, but even if it is not used, the algorithm can still remove all non-coalesced accesses and avoid duplications inside a thread block.

Second, when clustering is used, threads in different blocks may get swapped. However, unlike the padding algorithm case, even with the swapping, the sharing algorithm can still apply to a kernel containing multiple references with different access patterns. It is because the sharing algorithm does not require data references to remain or become regular. Consider the example mentioned earlier, $B[tid] + A[P[tid]]$. After clustering-incurred thread swapping, the references may become $B'[Q[tid]] + A'[P'[tid]]$ and both references become irregular. However, the second step of the sharing algorithm ensures that both arrays will be loaded into the shared memory in a coalesced manner. Accesses to the copies in the shared memory may be irregular, but recall that the performance of shared memory is resilient to access irregularity. It is worth noting that for this algorithm to work properly, the clustering and data reorganization need to put all the references ($B[tid]$ and $A[P[tid]]$ in our example) into consideration.

Third, the usage of shared memory may have two side effects on the performance of the kernel. The first is the time overhead of the introduced accesses to shared memory, which is often negligible compared to the time incurred by global memory accesses, especially for the irregular applications that are often memory latency bound. The second effect is that because shared memory is partitioned to all active thread blocks on a streaming multiprocessor, a large usage of shared memory by a thread block may reduce the number of thread blocks that can be active at the same time (called GPU *occupancy*.) Our experimental results in Section 5 show that the effect is not obvious on irregular applications.

Finally, when a problem size is large, the working set of a thread block could be larger than the shared memory. Fortunately, we observe that for most kernels, when the problem size increases, the problem size per thread block often remains unchanged but more thread blocks are created. In exceptional cases, to apply the sharing algorithm, the kernel can be modified to break the task of one block into smaller tasks and assign them to more thread blocks.

Analysis Quality-wise, as described in the algorithm, after the sharing algorithm applies, the accesses to the global memory become consecutive and coalesced. It maintains the zero non-coalesced accesses guaranteed by the duplication algorithm.

Space-wise, the algorithm saves space cost by avoiding data duplications for threads inside a block. The maximum number of copies of a data element is the number of thread blocks, rather than the number of threads in the duplication algorithm. If on average k ($k < B$, B for the number of threads per block) threads access one data element, with a perfect clustering that puts threads accessing the same data element into a single block, the algorithm can virtually avoid all data duplications. In practice, the amount of savings depends on the clustering quality (or how frequently multiple thread blocks access the same data elements if clustering is not used.) Section 5 provides the empirical results.

3.5 Discussion

The two new algorithms introduced in this section guarantee zero non-coalesced access as the duplication algorithm does. Although they reduce the space overhead of the duplication method substantially, it should be noted that they do not guarantee minimum space cost. Designing an algorithm with that guarantee and zero non-coalesced accesses is not the goal of this work. In fact, that task is no easier than the data positioning problem (they form dual problems with each other.) Section 5 will show that the two algorithms do provide practical solutions to a variety of programs.

4. Algorithm Selection and Integration

The three algorithms described in the previous section have different strengths and weaknesses. In this section, we provide a qualitative comparison, and describe an automatic selector and its integration in a runtime library.

Qualitative Comparison We summarize the qualitative differences among the three algorithms as follows.

```
// T: # of threads;   D: the set of memory references;
// $D_I$: working set of a thread block;
// Z: the size of the irregularly accessed data;
if T is less or comparable with Z:
    use duplication;
else if D has a single access pattern:
    use padding;
else if $D_I$ is smaller than shared memory:
    use sharing;
else:
    use duplication or change kernel to use sharing.
```

Figure 6. Guidelines for algorithm selection.

- **Applicability:** The padding algorithm is applicable to kernels that have a single reference pattern. While the other two algorithms do not have such a constraint, the sharing algorithm may need kernel modification when the working set of a thread block is too large to fit into shared memory, and the duplication algorithm may be applicable to only part of the data when the space limit is reached.
- **Space cost:** By avoiding unnecessary duplications, the padding and sharing algorithms use much less space than the duplication algorithm does.
- **Optimization capability:** All three algorithms have the capability to eliminate all non-coalesced memory accesses (in their applicable scenarios.) However, when being applied at runtime, the realizable benefits also depend on their runtime overhead.
- **Transformation overhead:** The time overhead of the duplication algorithm is in the creation and transfer of the new data copies, which can be substantial when the number of threads is very large or there are multiple references of different patterns to the same array. For the padding algorithm, the overhead includes the data and threads sorting time in addition to the creation and transfer of the new arrays. The overhead of the sharing algorithm consists of data creation and transfer time, the clustering time, the accesses to shared memory and the side effect on occupancy. Due to the large space reduction, the data creation and transfer in the two new algorithms usually take much less time than in the duplication algorithm. Data creation and transfer usually reside on the critical path of dynamic simulation applications, but the sorting and clustering in those two algorithms do not and hence can be largely hidden (e.g., through the CPU-GPU pipeline in G-Streamline [26].) We will come back to this point later in this section.

Algorithm Selection Based on the differences, we develop some simple guidelines, as Figure 6 shows, to help programmers select the suitable algorithm when writing a new program.

Meanwhile, we provide an automatic selector based on the online profiling and adaptive control offered by G-Streamline, a runtime library we previously developed [26]. The runtime library works when the GPU kernel is invoked in a loop. By profiling the initial several iterations during runtime along with some performance models of the system (e.g., the time to transfer a data from CPU to GPU, the time to create a data copy) built ahead of time through offline profiling, it estimates the kernel running time and optimization overhead to determine the suitable optimization algorithm to apply and the appropriate optimization parameters to use (e.g., the fraction of data to optimize in partial duplication.) Many irregular applications, including dynamic simulations and numerical calculations, are of that iterative pattern and are amenable for the runtime library to work. Our automatic selector employs the online profiling to estimate the amount of overhead of the algorithms and the kernel running time to pick the algorithm with the largest performance potential.

Integration with G-Streamline We integrate the selector and the reorganization algorithms into G-Streamline. G-Streamline uses a CPU-GPU pipelining scheme to allow runtime optimization of a future kernel invocation to happen on CPU when GPU is running the current invocation. However, if the future kernel's optimization depends on its previous invocation result, the optimization has to happen on the critical path. In that case, to make the optimization still happen asynchronously, kernel splitting is used so that the computations of a kernel are split and put into two parallel sub-kernels. The optimization of the second sub-kernel can run with the invocation of the first sub-kernel. The ratio between the amount of task between the second and first sub-kernel is called *transformation ratio*. The more costly the optimization is, the lower the ratio has to be so that the invocation of the first sub-kernel can hide the optimization overhead.

For all irregular applications we find, among the major operations in the three algorithms, sorting and clustering can happen across kernel invocations, but data creation and transfer are on the critical path due to dependences across kernel calls. They have to rely on kernel splitting to hide their overhead. In Section 5, we will see that the padding and sharing algorithms have much higher transformation ratio than duplication for their much smaller overhead in data creation and transfer. It is worth noting that G-Streamline uses its online profiling scheme to determine the suitable transformation ratio to ensure all overhead is hidden. If an optimization is infeasible to give benefits, G-Streamline shuts it down automatically to prevent any slowdown to the kernel.

Integrating the data reorganization algorithms into G-Streamline does not change the library's interface. It only adds a handful of functions as alternatives to the duplication algorithm already presenting in G-Streamline. The usage of the modified G-Streamline is the same as before [26]: Users insert several function calls into the GPU application to invoke the runtime asynchronous optimizations and online profiling; some minor changes to the kernel may be needed, including replacing old indexing arrays with new ones.

5. Evaluation

In this section, we evaluate the proposed algorithm assembly on eight benchmarks in Table 1, which all have dynamic irregular memory accesses. For comparing with the state of the art [26], they include all memory benchmarks used in the previous work: CFD is an unstructured grid finite volume solver; CG is a conjugate gradient method with sparse matrix-vector multiplication as its kernel; NN is for nearest-neighbor clustering; UNWRAP is for 3-D reconstruction. MD is a molecular dynamics simulation from the Shoc benchmark suite [7]; NBF and IRREG are derived from two irregular CPU applications heavily used by previous research [10]. The former is part of GROMOS, a force field simulation; the latter is the core of an iterative partial differential equation solver. The benchmark MERGE is a database update program. All code has gone through performance tuning to fit the execution models of GPU. The inputs to MD, IRREG, NBF and CFD consist of some nodes and neighbor lists generated randomly. The input to MERGE includes some indexing arrays of a set of data generated randomly. The inputs to CG contain a sparse matrix and vector. The locations of the non-zero elements in the matrix exhibit some patterns such that multiple rows of the matrix happen to multiple with a similar set of elements in the vector. The inputs to NN and UNWRAP come with the benchmarks.

We experiment on two types of GPU devices. One is NVIDIA Tesla C1060 hosted in a quad-core Intel Xeon E5640 machine, and the other is NVIDIA GTX480 hosted in a quad-core Intel Xeon E5520 machine. Both machines have CUDA4.2 installed. We obtain hardware performance through the NVIDIA's CUDA profiler.

Table 1. Benchmarks and selected optimization algorithms
(M1:Tesla C1060; M2: GTX480)

benchmark	description	alg. on M1	alg. on M2
MD	molecular dynamics	Sharing	Sharing
IRREG	partial diff. solver	Sharing	Sharing
NBF	force field	Sharing	Sharing
CFD	finite volume solver	Sharing	Sharing
CG	conjugate gradient	Sharing	Sharing
UNWRAP	3-D reconstruction	Dup.	(not runnable)
NN	nearest neighbor	Dup.	Dup.
MERGE	database update	Padding	Padding

unwrap cannot run on GTX480 for unknown reasons.

Table 2. Transformation ratios

benchmark	Dup.		Sharing	
	C1060	GTX480	C1060	GTX480
MD	0.25	0.1	0.85	0.65
IRREG	0.4	0.1	0.9	0.7
NBF	0.4	0.15	0.95	0.8
CFD	0.35	$*$	0.6	$*$
CG	0.45	0.15	0.5	0.2
UNWRAP	1	-	1	-
NN	0.7	0.4	0.7	0.4
MERGE	0.3	0.3	0.6	0.6

$*$: optimization is shut down; "-": not runnable.

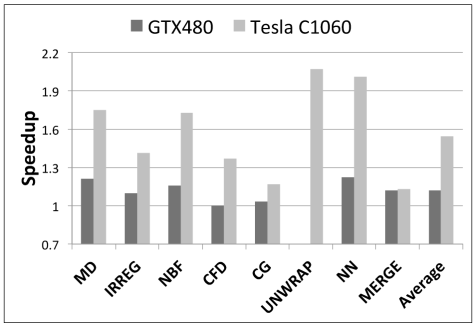


Figure 7. Speedup of selected algorithms

Results Overview Figure 7 reports the kernel speedups on both machines with the baseline as the execution time of the original GPU version. All overhead, including transformation and extra data transfer, is included. The selector-based algorithm assembly produces up to 21% speedup on GTX480. It gives even larger speedup (up to 109%) on C1060 because that device is more sensitive to irregular accesses for its lack of on-chip cache. (It is worth noting that having cache or not on massively parallel processors is still a debatable topic; some recent chips, such as Intel SCC, do not have cache for power efficiency.)

For further confirmation, we use the NVIDIA hardware performance profiler to measure the memory load efficiency. Load efficiency is defined as the ratio of requested global memory load throughput to actual global memory load throughput. As Figure 9

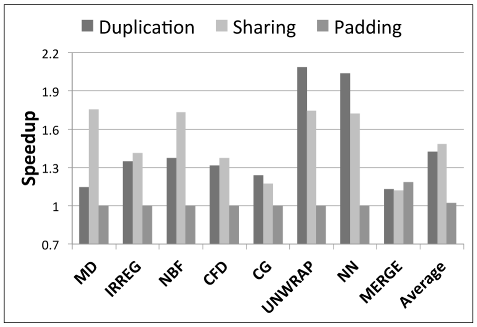


Figure 8. Speedup of all algorithms (Tesla C1060)

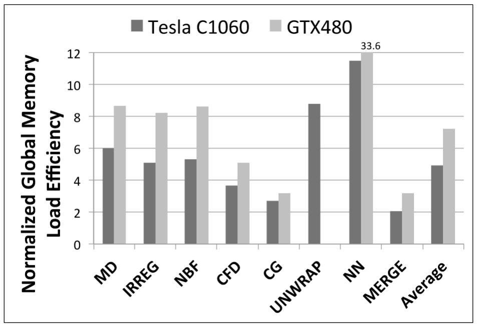


Figure 9. Memory load efficiency of selected algorithms.

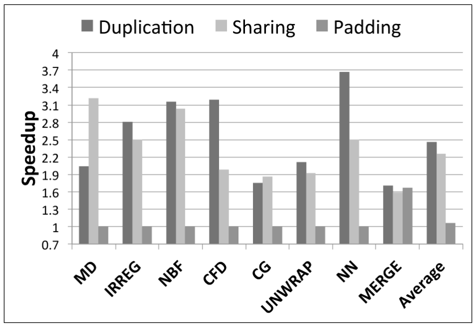


Figure 10. Potential speedups of all algorithms (Tesla C1060)

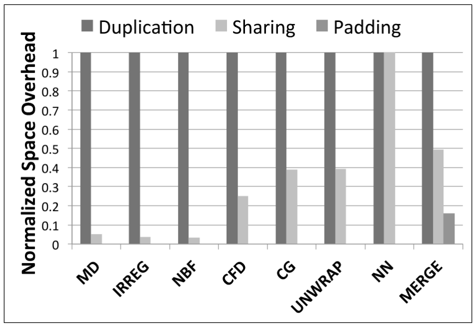


Figure 11. Normalized space overhead (*padding* is only applicable to MERGE.)

shows, the algorithm assembly improves the average efficiency by 4.9X on C1060 and 7.2X on GTX480 over the original version.

The two rightmost columns of Table 1 report the selected algorithms on the two machines. Figure 8 shows the speedups brought by each algorithm on Telsa C1060, confirming that all selections except for the one for CG on Telsa C1060 are correct. (We explain the selection error later in the detailed analysis on CG.)

Six of the eight benchmarks benefit the most from the newly designed algorithms on at least one machine. As Figure 8 shows, the new algorithms produce extra speedup as much as 8-60% over the duplication algorithm. It is mainly due to the larger transformation ratios (shown in Table 2) enabled by their large reduction of the overhead in data copy and transfer. The padding algorithm, due to its constraint on access patterns, is applicable to only the MERGE benchmark in the suite.

Overall, the results show that the two new algorithms significantly enhance the power of data reorganization for irregular memory optimizations. The algorithm assembly and online selector produce some promising speedups for most of the benchmarks. We next discuss each benchmark in further details.

MD, IRREG and NBF MD simulates the interactions of a number of molecules in a 3-D space. Two molecule nodes are neighbors if their distance is smaller than some predefined threshold. One thread is in charge of each node. In a simulation iteration, that thread traverses all its neighbors to calculate the force between each neighbor and that node. The inefficient memory references come from reading neighbors' positions.

The duplication algorithm improves the performance by duplicating position values to make sure adjacent threads load adjacent memory locations. Figure 10 shows that the full duplication can give 2X speedup on C1060 when overhead is not counted. But the overhead of data creation and transfer throttles the speedup to only 15%. The sharing algorithm has a higher performance potential than the duplication algorithm for the smaller working sets. Figure 11 reports that the sharing algorithm cuts the space overhead by 96%, which explains the seven times more speedup it creates than the duplication algorithm does when overhead is counted as Figure 7 shows.

The tremendous space reduction comes from two reasons. First, the irregular reference to data array is surrounded by a loop to traverse all neighbors, and in each iteration the memory access pattern of all threads is different depending on the topology of the interaction graph. The duplication algorithm, therefore, duplicates the same array the same number of times as the iteration number. Second, Sharing benefits greatly from clustering, which places adjacent nodes in topology closely in memory accessed by threads in the same block.

IRREG and NBF, like MD, have a graph topology. Figure 10 shows different potentials, because their kernels have different ratios of computation to memory accesses. Nonetheless, the sharing algorithm is also shown to be the best choice for them due to the reasons similar to MD. It is worth mentioning that the benefits of the optimizations also depend on the frequency of the neighbor list update in these simulation programs. When the update is frequent, the data transformations need to be applied often and hence lead to higher transformation overhead. When the overhead cannot be hidden completely, the runtime control of G-Streamline can adaptively adjust the fraction of data to transform to trade data layout quality for transformation efficiency [26]. A detailed study on various tradeoffs of the different algorithms in these frequent update scenarios are our future work.

CFD The program, CFD, computes force field of many particles. Each particle has substantially more features than the molecules in MD, and so each thread block processes more data. Figure 10 shows a potential of more than 3 times speedup from the duplication algorithm. But the data transfer overhead throttles the potential. The algorithm eventually produces 31% benefit with a 0.35

optimization ratio on C1060. The smaller space overhead of the sharing algorithm leads to a larger optimization ratio (0.6) and a higher speedup (37%.)

CG The kernel in CG does sparse matrix multiplication. The matrix is stored in the Compresses Sparse Rows (CSR) format. In the irregular kernel, one thread is in charge of one non-zero element in the sparse matrix. The accesses to the vector may not be coalesced depending on the sparsity in each row. The duplication and sharing perform similarly well in the potential graph. The best speedup on C1060 is 1.85 times, while the performance gain is around 20% on GTX480 due to the cache effects on the reuses of the elements in the vector. Figure 10 reports that the sharing algorithm has slightly larger potential than the duplication on C1060. The better algorithm, however, is duplication, because of the overhead caused by shared memory accesses. The subtle difference is not captured by the online algorithm selector, causing the sharing algorithm being selected. But the speedup lost is only less than 5%.

UNWRAP The kernel of this program is in a central loop, which transforms an image from the Cartesian coordinate system to the Polar coordinate system. Unlike the other programs, this program do not have data dependences carried by the different invocations of the kernel. The first tens of iterations of the program successfully overlap the overhead of both the duplication and sharing algorithms. The duplication was shown to be the better algorithm for its lack of the side effects in shared memory usage. (For unknown reasons, the benchmark cannot run on GTX480.)

NN The nearest neighbor identification program, NN, has a central loop to process an unstructured input file chunk by chunk. The kernel is launched once for each chunk, and calculates the Euclidean distances from the target location to each record in that chunk. At the end of the program, the main thread processes all distance results and obtains the K nearest neighbors. Figure 10 shows the large speedup potential on both C1060 and GTX480. The sharing algorithm does not reduce any space overhead as reported in Figure 11. The reason is that one record is only processed once, and the duplication algorithm essentially just transposes the data, creating no extra data copies. Like UNWRAP, there is no loop-carried dependence for NN, but the transformation and transfer overhead can not be fully overlapped, and we obtained 0.7 and 0.4 optimization ratios on C1060 and GTX480 respectively. On this special benchmark, the duplication algorithm is a better algorithm in both machines, producing higher speedup than sharing.

MERGE MERGE has the same access pattern for both loads and stores. Padding algorithm is applicable. As Figure 10 shows, Duplication and Padding have quite similar potential. Padding has a larger potential than Sharing because it needs no shared memory accesses. Padding reduces the size of transformed data significantly. Duplication, due to the memory size limit, only manages to transform 30% of data. Padding is the best choice on both GPUs for this program. The speedups on the two machines are quite similar on this program. The reason is that the program has few short reuse distances and hence does not benefit from cache much.

6. Related Work

Sections 1 and 5 have compared this work with previous studies [5, 22, 26] on optimizing dynamic irregular memory accesses on GPU. This section reviews some other related studies.

A number of studies have proposed compiler techniques to optimize GPU memory references. Examples include GPU optimizing compilers [12, 25], OpenMP-to-CUDA compilers [18], polyhedral models [2], performance tuning [20], and many others that cannot be listed for lack of space. All these techniques have focused on static irregularities that are amenable for compiler analysis. The usage of shared memory in the design of our sharing algorithm is inspired by some of those previous work [2]. But to our best knowledge, the sharing algorithm is the first algorithm that uses clustering and shared memory to avoid data duplications for runtime data reorganizations.

There are some recent studies exploring the synergistic usage of CPU and GPU, including the execution strategies proposed by Huo and others [13], the exploitation of OpenCL [16], and so on. In this work, we use the CPU-GPU pipeline created in G-Streamline [26] as it meets the needs for hiding transformation overhead.

Thread divergence is another type of dynamic irregularity in GPU, defined as the threads in a warp follow different paths of a kernel. Some hardware extensions have been proposed to remove thread divergences from a kernel execution [9, 19]. Carrillo and others [3] have proposed loop splitting and branch splitting to alleviate register pressure caused by diverging branches. As pointed out by an earlier work [26], thread divergence and non-coalesced memory accesses essentially stem from the similar source, a mismatch between threads and data. It suggests that the findings from this study are potentially usable for helping eliminate thread divergences as well.

Data reorganization has been used for many CPU data locality enhancements (e.g. [1, 4, 6, 8, 15, 24].) Some of them have especially concentrated on irregular applications [10, 21]. Kulkarni and others have studied locality issues of irregular data structures in the contexts of optimistic parallelism [17] and scheduling [14]. As a massively parallel architecture, GPU displays different memory access properties from CPU, exemplified by the hierarchical thread organizations, hardware enabled memory coalescing, and the SIMT execution model. All these features create differences in the challenges and opportunities in applying data reorganization, triggering the new set of innovations in this paper on both complexity analysis and transformation techniques.

7. Conclusion

This paper presents some fundamental understanding in exploiting data reorganization for minimizing non-coalesced memory accesses on GPU. It reveals the complexity of the problem by proving that it is NP-complete to create a data layout through data repositioning to minimize non-coalesced memory accesses on GPU, no matter whether thread reorganization is allowed. It points out that it is possible to circumvent the complexity by relaxing the space constraint in data repositioning. It introduces two new algorithms for minimizing non-coalesced memory accesses while avoiding the space inflation problem of a previous algorithm. It compares the various algorithms, presents some selection guidelines, and develops an automatic selector in a runtime library. Experiments show that the new algorithms excel previous techniques especially under space pressure. The algorithm assembly, assisted by the algorithm selector, enhances the performance of a set of dynamic irregular applications significantly, providing promising solutions to a large class of dynamic irregular references.

Acknowledgment

We thank Weizhen Mao for her suggestions to one of the NP-completeness proofs. We owe the anonymous reviewers our gratitude for their helpful comments to the paper. Some of the devices used in this study were donated by NVIDIA. This material is based upon work supported by the National Science Foundation under Grant No. 0811791 and CAREER Award, DOE Early Career Award, and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, DOE, or IBM.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006.

[2] 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*, pages 225–234, 2008.

[3] S. Carrillo, J. Siegel, and X. Li. A control-structure splitting optimization for gpgpu. In *Proceedings of ACM Computing Frontiers*, 2009.

[4] G. C. Cascaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 2000.

[5] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *SC*, 2011.

[6] T. M. Chilimbi and R. Shaham. Cache-conscious coallocation of hot data streams. In *PLDI*, 2006.

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

[8] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, 2004.

[9] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO'07*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[10] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel Distributed Systems*, 17(7):606–618, 2006.

[11] D. S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1995.

[12] A. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *ASPLOS*, 2011.

[13] X. Huo, V. Ravi, W. Ma, and G. Agrawal. An execution strategy and optimized runtime support for parallelizing irregular reductions on modern gpus. In *ICS*, 2011.

[14] Y. Jo and M. KulKarni. Enhancing locality for recursive traversals of recursive structures. In *OOPSLA*, 2011.

[15] M. Kandemir. A compiler technique for improving whole-program locality. In *POPL*, 2001.

[16] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. Opencl as a unified programming model for heterogeneous cpu/gpu clusters. In *PPoPP*, 2012.

[17] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS*, pages 233–243, 2008.

[18] S. Lee, S. Min, and R. Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. In *PPoPP*, 2009.

[19] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA*, 2010.

[20] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, pages 73–82, 2008.

[21] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *PLDI*, San Diego, CA, June 2003.

[22] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for simd cores. In *SC*, 2009.

[23] B. Wu, E. Zhang, and X. Shen. Enhancing data locality for dynamic simulations through asynchronous data transformations and adaptive control. In *PACT*, 2011.

[24] Y. Yan, X. Zhang, and Z. Zhang. Cacheminer: A runtime approach to exploit cache locality on *smp*. *IEEE Transactions on Parallel Distributed Systems*, 11(4):357–374, 2000.

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

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