Data Layout Optimization for GPU Programs

Yu Wang, Guanglin Xu
<yuw@andrew.cmu.edu>, <guanglix@andrew.cmu.edu>

Abstract
When targeting GPUs as the substrate, the performance of data-intensive applications are often sensitive to how well memory accesses coalesce. Existing techniques for coalescing optimization suffer two major drawbacks: 1. their limited applicability to only programs with *input-independent* array traversals and 2. The performance overhead caused by synchronizing duplicated data instances. In this paper, we propose a data layout optimization to minimize uncoalesced memory transfers via just-in-time array traversal profiling and multi-way graph partitioning. Our technique is applicable to optimize both input-dependent and independent array access patterns while requiring no duplication of data. Our results show that the proposed optimization is able to reduce uncoalesced memory transfers by up to 60% and reduces overall run-time by up to 32%.

1 Introduction

In recent years, *Graphics Processing Units* (*GPUs*) have gained much attention as an inexpensive solution for accelerating applications in various domains, such as machine learning and data mining. To exploit thread-level parallelism, modern GPUs organize software threads into groups known as *warps* [9] for execution in *SIMD* (*Single Instruction Multiple Data*) lock-step fashion. The off-chip *global memory*[1] on GPUs is divided into fixed length regions called *memory segments* [2]. During a SIMD load or store, the threads of a warp concurrently reference a set of memory locations, or *pattern*. If the addresses from a pattern point to the same memory segment, their accesses can be fully combined, or *coalesced*, as shown in Figure 1a. The spatial locality among concurrent SIMD accesses from threads in a warp, known as *Inter-Thread Locality* (*ITL*), determines coalescing performance. In this case, only a single data fetch is needed at presence of strong ITL. For a highly irregular memory access pattern, data references are diverged to different memory segments. Figure 1b demonstrates a case of poor ITL, where memory accesses are *uncoalesced* and have to be serviced by the memory controller separately.

Uncoalesced off-chip accesses are undesirable for incurrence of long latency due to serialization of multiple memory operations. The amount of memory transfers generated by a coalesced and an uncoalesced pattern can differ by up to a factor of *N*, the SIMD width of the platform (32 for Nvidia platforms and 64 for AMD platforms). In contrary to the abundance of computation resources, the relatively constrained memory performance of modern GPUs often can easily becomes a performance bottleneck, which characterizes the reduction of uncoalesced memory transfers an important optimization problem. Previous works have attempted to optimize coalescing from mainly two directions: static array indexing function analysis and data layout remapping. However, they support a limited subset of memory access behaviors. Static array indexing function analysis do not support irregular array traversal in which the indexing function is input data dependent. Existing data remapping techniques relies on heavy duplication of data elements, which incurs significant memory space overhead and requires costly synchronization of duplicated instances of a data element after its value has been updated. Applications such as common graph algorithms, in which the traversal pattern is data-dependent and vertex/edge data is updated, cannot be optimized.

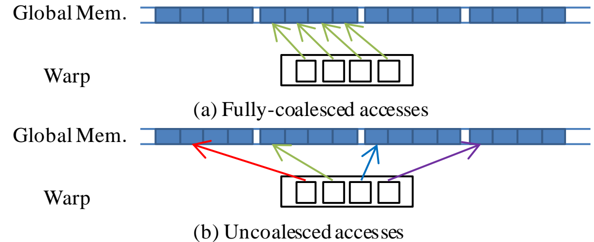


Figure 1. Examples of memory references with different degrees of inter-thread data locality. All figures assume 4 threads per warp and 4 objects per memory segment.

The goal of this work is to determine a data layout that minimizes uncoalesced memory accesses without duplicating data instances. As this problem has been theoretically proven to be NP-complete [2] and cannot be solved efficiently, our optimization technique, *Graph-Based Data Layout Remapping (GBDLR),* offers good solution approximation. To summarize our approach, we firstly collect pattern traces via *just-in-time* (*JIT*) profiling. The profiled traces are used to build a graph for capturing ITL among data objects. By defining the objective function as minimizing the sum of weighted edge cut, the problem of data layout remapping can be formulated as the standard problem of multi-way graph partitioning, and a number of existing heuristics can be applied to generate high quality solutions.

2 Technical Contributions

- We explore a new GPU program optimization technique, Graph-Based Data Layout Remapping. It maps the known NP-complete problem of re-

[1] **Note:** In this paper, the term memory is used for referring off-chip DRAM memory of GPU platforms, unless otherwise specified.

mapping array elements for coalescing enhancement to a well-studied graph partitioning problem, so existing partitioning heuristics can be exploited to generate high-quality solutions.

- To minimize optimization time, we explore two different graph formulations: 1. a hyper-graph formulation which is more precise and 2. a regular graph formulation which is more efficient to partition.
- We evaluate different options of graph partition heuristics.
- We implement a SIMD pattern profiler and propose a technique called *selective pattern sampling,* to reduce the profiling and graph partitioning time.
- We evaluate our proposed optimization scheme using a sparse-matrix vector multiplication GPU kernel with multiple sparse matrices as input data.

3 Related Work

In this section, we will examine some representative works focus to reduce uncoalesced memory accesses for GPU applications. For the purpose of example illustration, memory references to an array *A* is symbolized as *A[idx()],* where *idx()* is the array indexing function, the length of A = *|A|* ,and the size of an array element =*d.* For simplicity, we assume *warp size = w, memory segment size =* $w \times d$, and thread index = *tid.*

3.1 Data Layout Remapping

Data Layout Remapping (DLR) rearranges the elements of *A* to *|A|/w* segments, such that total degree of inter-segment span of SIMD patterns are minimized. This is a trivial problem when each array element is accessed no more than once, as shown in Figure 2a. In this case, the members of the *Nth* segment would simply be *{A[idx(n)] | n⊆tid from Nth warp}*. However, SIMD patterns often overlap and the overlapped array elements are accessed multiple times, as illustrated in Figure 2b. When overlapping happens, DLR has been shown to be NP-Complete [2], and finding the global optimal data layout is impractical.

In an effort to tackle the complexity of general DLR, a simple strategy called *duplication* was proposed by Zhang et al. [4]. As illustrated in Figure 2c, the duplication algorithm creates a new array *A'* such that *A'[tid]=A[fcn(tid)]*, and then all references to *A[fcn(tid)]* in the program are replaced with *A'[tid]*. As array *A'* is sized to match the number of threads in a program, the duplication algorithm guarantees zero non-coalesced memory access at the cost of large memory footprint overhead. To reduce storage space requirement, Wu et al. [2] proposes the *padding* and *sharing* techniques to reduce the degree of data duplication by conditionally merging duplicated segments.

3.2 Optimizations for Static Array Indexing Functions

Several static coalescing optimization techniques were proposed for GPU programs with input-independent array indexing functions [3][5]. The gist of these techniques are similar: classify array indexing function and then apply corresponding code transforms for optimization. Yang et

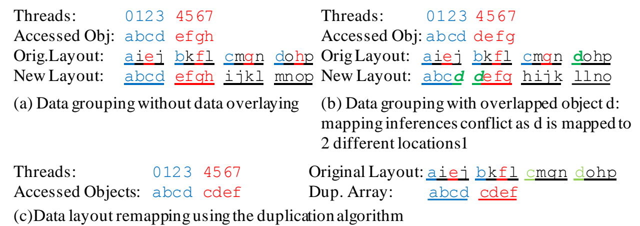


Figure 2. Examples to illustrate the data grouping problem with/ without data overlaying and the duplication algorithm

al. suggest pre-fetching blocks of matrix data to on-chip memory in a coalesced fashion to avoid uncoalesced row-traversals in matrix multiplication [3]. Cuda-Lite [3] transforms the order of nested loops to force warp threads traverse a matrix in column-wise direction so SIMD accesses can be coalesced perfectly. Optimizations in this category are efficient as they are fully static and require no run-time data, but their scope is limited to the memory accesses that involves no indirection.

4 Graph-Based Data Layout Remapping

The fundamental issue of Data Layout Remapping is its complexity. Prior works get around solving this problem by creating duplicated data instances. As pointed out in section 1, this incurs overhead on both memory space and performance. In this work, we tackle the complexity by interpreting DLR as a well studied graph problem in order to facilitate existing heuristics for approximating the solutions. In this section, we'll examine the two different formulations we proposed for DLR: a more precise hypergraph formulation and a standard graph formulation which can be more efficient to solve.

4.1 Hypergraph Formulation

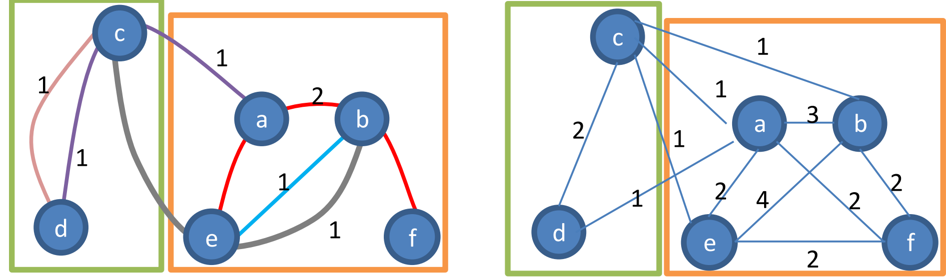
Given a trace of SIMD patterns, the first step of GBDLR is to represent ITL relationship among accessed data elements as a ***co-occurrence graph,*** an undirected and edge-weighted hypergraph. In a co-occurrence graph, a vertex represents an accessed data element, and a hyperedge spans across data elements appearing in its corresponding pattern. The weight of a hyperedge represents the number of times a SIMD pattern appears in a trace. If a set of data addresses appear in many pattern, or they *co-occur* frequently, the sum of weights for hyperedges that connect them will be large, which indicates them should be mapped to the same memory segment. Given a GPU kernel, its input data and launching configuration, including the number of threads and warp size, the pattern trace can be collected using a JIT profiler, which will be discuss more in section 5.

Given a co-occurrence graph, DLR can cast into a standard *k-way graph partitioning problem* by defining *k=|A|/w* and *the* cost function to be $\sum(|e|)$ *| e⊆all edges of the graph,* where |e|=number of partitions *e* spans to multiplied by its weight. Solving this problem is equivalent to partitioning the accessed array elements to *|A|/w segments,* while minimizing the cost. Since *cost + number of patterns* is equivalent to the total number of segment transfers, cost minimization is same as minimizing uncoalesced access. As a result this hypergraph formulation is precisely equivalent to the original definition of DLR.

Threads: 0123 4567
SIMD mem access1: bbbe abef
SIMD mem access2: cdca abef
SIMD mem access3: cdcd bbec

Original Layout: aceg bfdh

GBDLR layout: abef cdgh



(a) Hypergraph formulation

(b) Regular graph formulation

Figure 3. An example of GBDLR
The original layout requires 12 memory accesses. As both the hypergraph and the regular graph formulations produce the same partitioning result, they produced the same data layout , which requires only 8 segment transfers. Assume 4 threads per warp and 4 objects per memory segment.

Figure 3a shows the co-occurrence graph and the data remapping result using our hypergraph formulation for a simple trace of 6 patterns. Note the partitioning cost (2) plus the number of patterns (6) is equivalent the total memory segment transfer (8).

4.2 Standard Graph Formulation

Although the hypergraph version of GBDLR guarantees a precise mapping to the original DLR definition, hypergraph partitioning is often more expensive than standard graph partitioning. For input-data-dependent array traversals, GBDLR cannot be performed until SIMD pattern traces are generated by JIT profiling, which makes the performance of graph partitioning important. This fact motivates us to reformulate GBDLR using standard graphs.

When using standard graph formulation, a ***co-occurrence graph,*** is a complete, undirected and edge-weighted graph. Each edge connecting two vertices is weighted by their *co-occurrence frequency*, where co-occurrence is defined as an instance that two array elements simultaneously occur in the same SIMD pattern. Co-occurrence frequency measures how often two array elements co-occur and captures as potential performance benefit if two data elements are mapped to the same memory segment. If two elements never co-occur, the edge connecting them is weighted by zero.

With the new co-occurence graph definition, DLR can be approximated with the same k-way graph partitioning from the hypergraph formulation. The only change required is to redefine the cost function as the sum of edge cut weights. This specific type of k-way graph partitioning is known as *Minimum K-Cut* [10]. Minimum K-Cut is an important problem in domains such as circuit placement, and many known heuristics can be employed to generate high quality solutions.

Figure 3b shows the co-occurrence graph and the data remapping result using our standard graph formulation for the same trace used in figure 3a. It produces the same layout mapping in this specific case. However, as the formulation is less precise, the final cost cannot be directly used to calculate the number of segment transfers.

5 SIMD Pattern Profiler

5.1 Requirements

As a precondition to optimize array access of an OpenCL program, a profiler extracts SIMD patterns from the source code. In this section, we'll use an example to illustrate the requirement of our profiler. SIMD patterns are divided into two categories. Patterns in the first category can be determined in compile time because relevant information is ready before running the program. The accesses to *mrp* in Figure 4 correspond to one such case. On the other hand, patterns in the second category depend on input data and couldn't be determined until runtime. The accesses to *A* in Figure 4 belong to the latter category.

```
1:    unsigned int row = get_global_id(0);
2:    if ( row < m ) {
3:      float sum = 0;
4:      for (unsigned int i = 0; i < matrixRowSize[row]; i++) {
5:        unsigned int j = mrp[row] + i;
6:        sum += A[j] * vectorB[ adj[j] ];
7:      }
8:      resultVector[row] = sum;
9:    }
```

Figure 4: OpenCL kernel of sparse matrix vector multiplication

The SIMD patterns of *mrp* and *A* which our profiler aims to harvest are shown in table 1 and table 2 respectively. In both tables, each column represents array accesses for a *thread-id*, while each row represents array accesses for a *loop-id*. Since the index of *mrp* points to the *thread-id* (by returning from *get_global_id(0)*), it's value increments along the thread space horizontally and keeps unchanged along the iteration space vertically. For *A*, the index is indirect and requires values stored in array *mrp*.

Table 1: SIMD Patterns of *mrp* in a warp

	Thread 0	Thread 1	…	Thread w[2]
Loop 0	0	1	…	w
Loop 1	0	1	…	w
…	…	…	…	…
Loop n	0	1	…	w

[2] w refers to the warp size. It is 32 in Nvidia devices and 64 in AMD devices.

Table 2: SIMD Patterns of *A* in a warp

	Thread 0	Thread 1	…	Thread w
Loop 0	mrp[0]+0	mrp[1]+0	…	mrp[w]+0
Loop 1	mrp[0]+1	mrp[1]+1	…	mrp[w]+1
…	…	…	…	…
Loop n	mrp[0]+n	mrp[1]+n	…	mrp[w]+n

5.2 Profiler Design

Our profiler is based on the Clang tooling framework. Clang exposes C++ classes to read the Abstract Syntax Tree (AST) of OpenCL kernel. AST is a structural representation of source code, which facilitates extracting useful information relevant with SIMD patterns.

Information extraction via AST boils down to walking along the syntax tree. For example, once an array access occurs as an *ArraySubscriptExpr* node in the AST, the base name of the array can be retrieved via visiting an ImplicitCastExpr node followed by a DeclRefExpr node. It is trivial to detect if an array access is a write or not by checking if this access is in the LHS of a "=", "+=", "-=", "*=" or "/=" operator node. However, calculating the value of array index is non-trivial. For indices that can be determined in compile time, they usually require the use of *thread-id* and *loop-id*. For other array indexing schemes that rely on runtime data, knowledge from additional input in runtime is inevitable, and the patterns has to be collected in JIT fashion.

5.3 Optimizing Profiler

For parsing in compile time, performance is less concerned. However, the performance of JIT profiling is critical. In the following sections, two kinds of optimizations are introduced to reduce the execution time of the profiler.

5.3.1 General Optimization

With a naïve profiler implementation, calculating the value of an array index always requires walking through the AST. Its overhead can be significant when the number of nodes being walked is large. For those array indices whose value is independent with *loop-id*, the value can be stored in a lookup table so future calculation for obtaining the same value can be eliminated. Intermediate results relevant to the value can also be stored at a lookup table. Currently we have employed value lookup tables for kernel function parameters, *thread-id*, for loop parent node and array base name. They reduce 65% of the runtime in our amazon benchmark.

5.3.2 Domain-specific Optimization

Domain-specific optimization is achieved by *selectively sampling* array indices. By omitting less important array indices, it reduces not only the execution time of the profiler but also the inference graph size for graph-based DLM. Considering the length of SIMD pattern as the number of array accesses in that pattern, a SIMD pattern with smaller length is called a "scatter pattern" while that with larger length is called a "dense pattern". In the context of optimizing memory access of uncoalesced SIMD patterns, a scatter pattern contains fewer memory addresses and the degree of inter-segment accesses is likely to be low even without optimization. As a result, omitting scatter patterns has minor impact on the quality of the final data layout. While it is hard to predict SIMD patterns for general programs, patterns from some domain program can be predictable. In the example of Figure 4, some threads can terminate relatively early to its neighboring threads from the same warp due to the differences of the row size values, so SIMD pattern lengths often decreases when *loop-id* increases as warp divergence happens. To exploit this fact for reducing the size of generated pattern traces, we employ a technique called *selective sampling* which discards SIMD patterns with lengths under certain thresholds. This approach effectively filters short patterns from the ending loop iterations and improves run-time performances for our JIT profiler as well as data layout optimizer.

6 Evaluation

Table 3 summarizes the methodology used to evaluate our proposed coalescing optimization techniques. We choose Sparse Matrix Vector Multiplication as our GPU benchmark as graph structures are often stored in sparse matrix format, so the evaluation results is representative general graph algorithms such as pageRank. We choose 3 standard sparse matrix datasets with different sizes and sparseness for our experiments. We also perform study on how selective pattern sampling will impact the data partitioning and GPU programs.

Table 3: Methodology

Platform for pattern profiling	Intel i7 4770k 32GB DDR3 DRAM
Platform for graph partitioning and GBDLR performance evaluation	Intel i7 3730qm 2.40 GHz 16GB DDR3 DRAM Nvidia GTX 680m
GPU benchmark	OpenCL Sparse-matrix-vector multiplication
Input data	3 Standard sparse matrices: 1. erdos-10k: 10k vertices 2. bara-40k: 40k vertices 3. Amazon: 3 million vertices
Selective pattern sampling configuration	Tested 2 thresholds: 1.Filter pattern length less than 4 2.Filter pattern length less than 16
Graph partitioner	Hypergraph: hMetis Standard graph: Kernighan-Lin (KL) and Fiduccia-Mattheyses-Sanchis (FMS)

6.1 Effectiveness of GBDLR

Figure 5.b demonstrates the effectiveness of GBDLR on reducing uncoalesced accesses. Our KL implementation cannot finish partitioning the dataset Amazon before the project deadline, thus the result is not shown. On average,

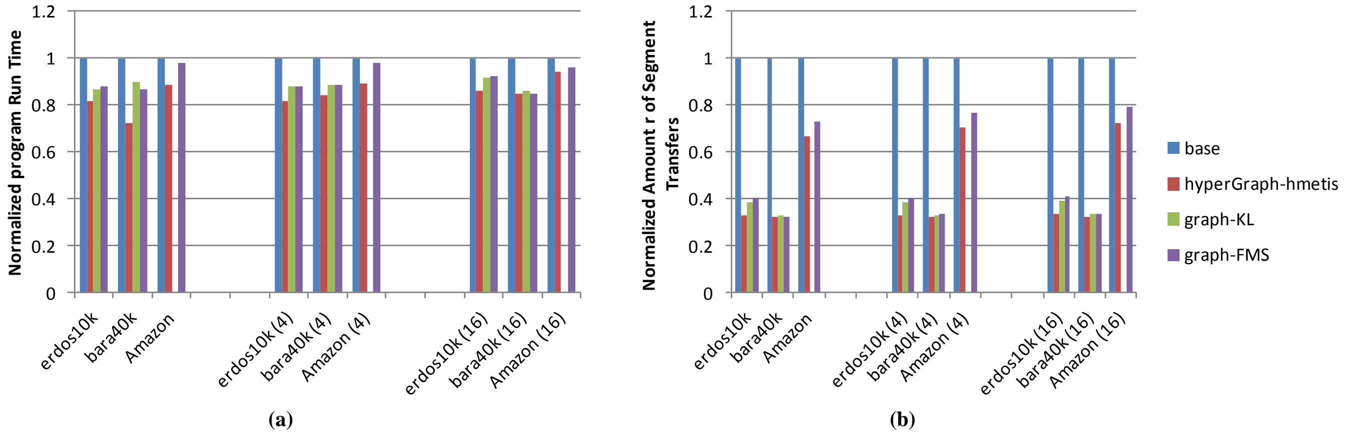


Figure 5. (a) Normalized run-time and (b) Normalized memory transfers for sparse matrix vector multiplication with GBDLR optimization

with hypergraph formulation, the amount of memory segment transfers are reduced by 58% across the three input datasets. With standard graph formulation, the GBDLR performs slightly worse as it reduces segment transfers by an averaged 51%. Figure 5.a shows the runtime of Sparse Matrix Vector Multiplication GPU kernel after GBDLR compare to the un-optimized baseline. On average, GBDLR delivers 16% and 11% speedups for hypergraph and standard graph respectively.

6.2 Practicality and Future work

Despite that GBDLR successfully demonstrates its capability to improve GPU coalescing performance, the partitioning of Co-Occurrence Graph turns out to be a major issue. For erdos10k and bara40k datasets, KL took 543 milliseconds and 6.7 seconds to complete partitioning respectively, while the GPU kernel only take couple milliseconds to execute. The FMS practitioner is faster by approximately a factor of 2, but it still much slower than the GPU program run-time. hMetis for hypergraph partitioning runs even slower than KL across all test cases. For the Amazon dataset, it took almost three hours to complete the partitioning tasks. Selective Pattern Sampling with threshold of 16 does improve the partitioning time by 44% in this case, but it is still too slow to for the standard of JIT optimization. Fortunately, there are graph algorithms that requires repetitively executing the same kernel until the results stabilizes, such as pageRank and 3D stereo matching, which amortizes the expensive optimization of GBDLR with long program run-time. Until we study those cases, it is still too early to judge the effectiveness of our optimization approach.

7 Conclusion

Reducing uncoalesced data references is critical for the performance of memory-intensive GPU applications. However, existing coalescing optimizations cover a very limited subset of memory access behaviors. To address this problem, we propose Graph-Based Data Layout Remapping. By mapping the NP-complete problem of rearranging array elements to minimize uncoalesced memory references to the standard K-way graph portioning problem, known heuristics for solution approximation can be facilitated. Our evaluation suggests that GBDLR can indeed improve coalescing performance for GPU programs. However, the heavy overhead of graph partitioning prevents this technique to be use for JIT optimization.

References

[1] E. Enurvitadhi et al. “Compiling Graph Algorithms for Accelerator Platforms,” submitted to *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[2] B. Wu et al. “Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-Coalesced Memory Accesses on GPU,” in *Principles and Practice of Parallel Programming (PPOP)*, 2013.

[3] Y. Yang et al. “A GPGPU Compiler for Memory Optimization and Parallelism Management,” in *Programming Language Design and Implementation (PLDI),* 2010.

[4] E. Zhang et al. “On-the-fly elimination of dynamic irregularities for gpu computing,” *in Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2011.

[5] S. Ueng et al. “CUDA-Lite: Reducing GPU Programming complexity,” *in Languages and Compilers for Parallel Computing (LCPC),* 2008

[6] S. Che, “Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems,” *in High Performance Computing, Networking, Storage and Analysis,* 2011.

[7] H. Eslami et al. “A GPU Implementation of Tiled Belief Propagation on Markov Random Fields,” in *MEMOCODE,* 2013.

[8] C. Schulz, *High Quality Graph Partitioning*. Berlin, Germany, epubli GmbH 2013, pp. 39-41.

[9] Nvidia Co. "Nvidia OpenCL Programming Guide, " http://www.nvidia.com/cotent/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf, 2009

[10] W. Kernighan et al., "An efficient heuristic procedure for partitioning graphs," *Bell Systems Technical Journal* 49, 1970, Pages 291–307

Contributions:

Yu Wang 50%

Guanglin Xu 50%