Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems

Shuai Che, Jeremy W. Sheaffer and Kevin Skadron
{sc5nf, jws9c, skadron}@virginia.edu

Department of Computer Science, University of Virginia

ABSTRACT

Graphics processors (GPUs) have emerged as an important platform for general purpose computing. GPUs offer a large number of parallel cores and have access to high memory bandwidth; however, data structure layouts in GPU memory often lead to sub-optimal performance for programs designed with a CPU memory interface—or no particular memory interface at all!—in mind. This implies that application performance is highly sensitive irregularity in memory access patterns. This issue is all the more important due to the growing disparity between core and DRAM clocks; memory interfaces have increasingly become bottlenecks in computer systems.

In this paper, we propose a simple API, *Dymaxion*[1], that allows programmers to optimize memory mappings to improve the efficiency of memory accesses on heterogeneous platforms. Use of Dymaxion requires only minimal modifications to existing CUDA programs. Our current framework extends NVIDIA's CUDA API with the addition of memory layout remapping and index transformation. We consider the overhead of layout remapping and effectively hide it through chunking and overlapping with PCI-E transfer. We present the implementation of Dymaxion and its optimizations and evaluate a variety of important memory access patterns. Using four case studies, we are able to achieve $3.3\times$ speedup on GPU kernels and 20% overall performance improvement, including the PCI-E transfer, over the original CUDA implementations on an NVIDIA GTX 480 GPU. We also explore the importance of maintaining per-device data layouts and cross-device data mappings with a case study of concurrent CPU-GPU execution.

Categories and Subject Descriptors: C.0 [Computer Systems Organization]: hardware/software interfaces

General Terms: Performance, Measurement

Keywords: Heterogeneous Computer Architectures, GPGPU, Memory Access and Data Layout, Latency Hiding

1. INTRODUCTION

Memory bandwidth and latency present serious concerns that limit throughput in multicore and manycore architectures. These challenges are getting worse, as the number of processing elements per chip is growing much faster than bandwidth and latency are improving. This problem is particularly acute in GPUs, because of their wide memory interfaces and SIMD [13,16] organization. Furthermore, their performance relies on effective memory bandwidth utilization [9,20,21].

[1]Our choice of this name is inspired by a Dymaxion map, which is a projection of the world map onto the surface of a polyhedron, that can be flattened in various ways to form a 2-D map.

An application's algorithmic behavior, as viewed by the programmer, does not necessarily lead to the most efficient memory access pattern. Today's GPU programming models require programmers to invest considerable manual effort to optimize memory accesses for high performance. For instance, GPUs' SIMD architectures require efficient memory coalescing for inter-thread data locality. Hybrid memory units—such as the GPU's *shared*, *constant*, and *texture* memories—present access patterns that are unfamiliar and unintuitive to programmers and that favor specific, specialized mappings. However, code optimized for specialized access patterns may not perform well or be portable across multiple vendors' platforms and different hardware generations. Additionally, for efficient heterogeneous computing, different architectures and multithreading models may favor different memory mappings. For example, SIMD organizations generally perform best when each thread or lane of a SIMD operation accesses adjacent data, while scalar organizations perform best when a single thread accesses adjacent data. This in turn requires heterogeneity in data layout as well as per-device optimization for simultaneous execution.

This paper addresses these concerns with a set of software-level abstractions, APIs, and underlying mechanisms to ease programmer burden while improving memory access efficiency in unoptimized code. Dymaxion currently targets GPUs, but can be targeted to any platform. Dymaxion is also helpful for increasing the efficiency at each node for high performance computing, given the growing use of GPUs. For instance, as with any GPU cluster, an MPI process launched on each node can make use of GPUs by making CUDA calls. We find that optimizing access patterns yields substantial performance improvement by effectively hiding memory remapping latency.

This work makes the following contributions:

- We present an API framework and a data index transformation mechanism that allows users to reorganize the layout of data structures such that they are amenable to localized, contiguous access by simultaneous GPU threads. In CUDA, this implies that thread accesses can be coalesced efficiently.

- We show how to hide the overhead of layout remapping during PCI-E transfer, taking advantage of simultaneous CUDA streams. Memory layout transformation is divided into separate chunks and overlaps with PCI-E memory transfer. We also compare it to a technique which takes advantage of the *zero copy* feature on the GPU.

- We evaluate several representative access patterns common in many scientific applications and use several case studies to present the use of our framework in achieving better coupling of access patterns and actual memory layouts.

- We present a case study of spreading work simultaneously across the CPU and the GPU, in which the two platforms prefer different mappings of data layouts and access patterns respectively. We show that our framework is a clean abstraction and convenient software-level building block to ensure cross-device data coherency.

An API-based remapping mechanism has the benefit of giving programmers more control and flexibility over data mapping. Dymaxion allows hints to be provided to the system to influence memory mapping based on programmers' knowledge of the algorithms.

Another advantage of an API-based approach is portability across platforms, as an API can be optimized for different architectures. We develop Dymaxion as an extension to NVIDIA's CUDA; however, the same framework can be extended to other GPU or heterogeneous programming models, such as OpenCL [18]. Four diverse applications from the Rodinia suite are used in our evaluation [3]. Using Dymaxion on a GTX 480 GPU, an average of $3.3\times$ speedup is achieved on compute kernels and a 20% performance improvement is achieved, including the PCI-E transfer, when compared with their original CUDA implementations. Additionally, the extra programming effort involved in using Dymaxion is trivial.

2. MOTIVATION

The impetus for Dymaxion lies in three key observations, discussed here.

2.1 CUDA Coalescing

One important performance optimization for GPUs (supported on NVIDIA hardware starting with the GT200 generation) is the coalescing of global memory accesses generated by streaming multiprocessors (SMs). The SMs schedule and execute threads in lockstep groups of 32 threads called *warps*. Global memory accesses within a half-warp will be coalesced into the minimum number of memory transactions [7]. Figure 1 shows a simple example: if the k^{th} thread accesses the k^{th} word in a segment, a single 64-byte transaction is required. Different scenarios and requirements for memory coalescing are documented in detail in the NVIDIA technical guides [6, 7].

2.2 Memory Locality of Inter-thread Accesses

The following code segment shows two simple examples of CUDA code that loop over the data elements of a 2-D array and assigns their values to another array. In each iteration, a strip of data elements is accessed concurrently.

```
/* The CUDA implementation               */
int bx = blockIdx.x;  /* thread block ID */
int tx = threadIdx.x; /* thread ID       */
int tid = BLOCK_SIZE * bx + tx;

//Example 1: access different rows (row-major)
for (i = 0; i < N; i++) {
        des[cols * tid + i] = src[cols * tid + i];
}
//Example 2: access contiguous data elements (column-major)
for (i = 0; i < N; i++) {
        des[cols * i + tid] = src[cols * i + tid];
}
```

In this implementation, the accesses are parallelized, each thread responsible for processing one element. One important observation is that if the thread id, `tid`, is used as the lowest dimension of the index to access an array, as in `array[cols * i + tid]` (See Example 2), multiple simultaneous threads will access contiguous memory locations. Thus, the memory accesses of the second loop manifest better inter-thread spatial locality than those of the first. In fact, the need for both types of accesses comes up in many applications (e.g. matrix multiplication).

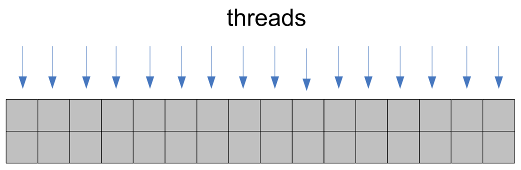

Figure 1: The memory coalescing concept. If threads access contiguous data elements, multiple thread accesses can be coalesced. Each element is 4 bytes in this example.

An example shows how poor locality of concurrent memory accesses leads to poor performance and scalability of GPU applications. Figure 2 shows the performance of two versions of a k-*means* GPU implementation, which we will discuss in Section 5.1 in more details. One version assigns each thread to compute a row of the main data structure, which is row-major. In this version, each row represents a data element while each column represents a feature; different data elements can be processed in parallel. This organization results in suboptimal memory coalescing for threads within a warp. In contrast, the other implementation uses a column-major layout, in which threads within a warp access adjacent data elements and achieve better inter-thread locality. We vary the number of features in the main data structure and measure execution times on NVIDIA GeForce GTX 480 and 285 GPUs.

The column-major organization achieves better performance on both platforms. But when the number of features surpasses 16, the 480, a more powerful GPU running a row-major based k-*means*, actually yields poorer performance than the 285; this application is memory-bound and benefits from coalescing, which requires a column-major organization so that warps access contiguous data. This example illustrates how much impact memory access patterns play in GPU performance. To solve this issue, Dymaxion enables programmers to match memory access patterns and data layouts automatically through a simple programming interface that declares the access pattern in terms of the original data structure. This high-level knowledge then permits transparent memory layout remapping to optimize bandwidth (assuming that all accesses to the data structure are mediated by the API).

On the other hand, CPUs and GPUs may prefer different data layouts for certain applications. This is due to the fact that better cache locality is needed for contiguous memory accesses issued by individual CPU threads, while an efficient GPU memory transaction is desirable to feed data to multiple simultaneous SIMD

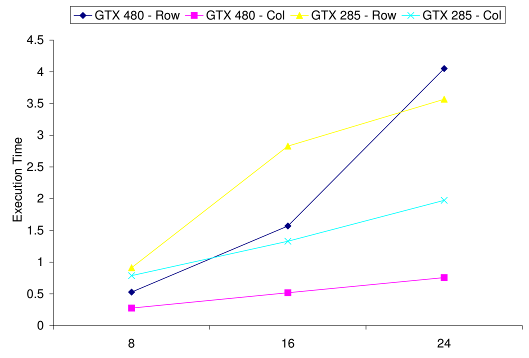

Figure 2: The x-axis represents feature size while the y-axis represents execution time. Execution time is one iteration of the k-*means* distance kernel with an input of 64 k data objects. More features mean more uncoalesced memory accesses.

Table 1: **Fraction of total execution time devoted to PCI-E transfers**

Applications	PCI-E Transfer	GPU Kernel
K-means	51%	49%
Needleman-Wunsch (NW)	32%	68%
SpMV	77%	23%
Nearest Neighbor (NN)	70%	30%

threads. For instance, for the same *k-means* problem, CPUs, in contrast, favor a row-major layout, as we discuss in Section 6. Furthermore, in contrast to the CPU, GPU has a distinct memory hierarchy with specialized memories, each of which prefers a different mapping between data layout and access pattern. For example, a typical texture unit design adopts a Morton-curve access pattern. An efficient use of constant memory requires simultaneous thread accesses from a single warp to touch the same cache lines; therefore, hand-optimizing memory mappings for different platforms is not only tedious, but also the relevant code may need rewritten for good performance and portability across platforms. To resolve these issues, we need a high-level abstraction to define memory mappings.

2.3 Making Data Ready on the GPU During PCI-E transfer

Often GPU applications expend significant time on data transfer between system and GPU device memory [5, 7]. Because PCI-E transfers have less available bandwidth than DRAM accesses, and also because of the device call overhead associated with each transfer, an efficient implementation should minimize data transfer, both instances and volume.

Table 1 shows the fractions of total execution time dedicated to CPU-GPU memory transfer and GPU kernel execution. PCI-E transfers consume a large fraction of execution time in all four applications. In order to both leverage and reduce this overhead, we propose that additional functionality, such as memory remapping, be implemented during PCI-E communication in order to increase memory locality for subsequent GPU computation. Such functionality can be implemented either in software (the driver), hardware (the DMA mechanism), or both. Ideally, these operations would be programmable to maximize their generality. Because we do not have access to proprietary GPU drivers, our prototype Dymaxion implementation takes advantage of CUDA stream functionality to aid data reorganization. Furthermore, the data structure is also broken into chunks to hide the latency of memory remapping, described further in the next section.

3. DYMAXION DESIGN AND IMPLEMENTATION

In this section we describe the design and implementation of Dymaxion.

3.1 CPU-GPU Data Transfer and Remapping

In our framework, programmers start by calling remapping functions on the target data structures. This launches a series of operations to transfer data between the CPU and the GPU in a remapped order that yields efficient data access for the GPU compute kernel (see Figure 3). The memory layout transformation brings new overhead, which we attempt to minimize while trading it off for improvements in GPU data locality. If the reorganization overhead is less than the time required for PCI-E transfer, most of it can be hidden through pipelining. We found that CPU exhibits lower

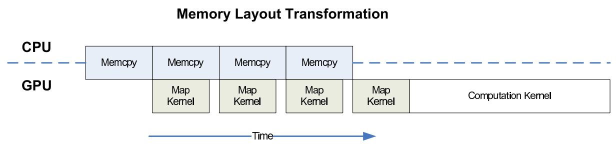

Figure 3: Memory layout reorganization. The entire data structure is broken into small chunks and transfered from the CPU and the GPU chunk by chunk. After each chunk completes transfer, layout reorganization is applied to that particular chunk on the GPU. We assume *memcpy* is executed sequentially.

bandwidth than the PCI-E transfers, so we decided to take advantage of the high bandwidth and deep multithreading features of the GPU for layout reorganization.

Currently, the remapping flow can be broken down into two major components:

1. Break the data into small chunks and transfer each chunk asynchronously from the CPU to the GPU one by one.

2. Immediately after data transfer of each chunk, launch remapping kernels to reorganize data layout; each thread is responsible for relocating one data element.

Figure 3 illustrates the idea of overlapping PCI-E transfer and layout transformation on the GPU. In this example, we assume the PCI-E implementation is serial. In reality, it can be implemented more efficiently using parallelism. However, we are missing implementation details necessary to take advantage of that organization. Because there is a one-to-one mapping between the original and remapped locations, and because the remappings do not overlap, the remapping of data are independent; however, the latter invariant introduces storage overhead, mitigated by chunking.

Figure 4 shows sample code illustrating one possible implementation with CUDA streams. CUDA applications often manage concurrency through streams [7]. A stream, in CUDA, is a sequence of commands that execute in order. Distinct streams are only partially ordered [7]. To use streams, we allocate our host memories with `cudaMallocHost()`. This example is similar to the stream example in the CUDA programming guide [7], the difference being that, for each CUDA stream, `stream[i]`, we first copy a chunk and then execute a specific kernel to perform layout remapping for that chunk. The chunk index `i` and chunk size are used to determine which data to reorganize. In Section 5.7, we compare this approach with an alternative one using the *zero-copy* feature.

Note that data reorganization is one of the implementation options for memory remapping in Dymaxion, designed as a high-level abstraction. Other possible approaches include physical-address-to-physical-address translations and associated latency hiding techniques [22]. Another possibility is to leverage MMU for layout transformation, avoiding the extra CPU-GPU copy, but that doing the transformation across the PCI-E hub may not be as efficient as a bulk copy onto the GPU card, from which the GPU's massive bandwidth and parallelism can be leveraged to speed up the transformation.

3.2 Index Transformation

Following the layout transformation, a GPU device memory pointer for the reorganized data structure is returned for the user to pass to the compute kernel. Because of the change in layout, the indices of future accesses must also be transformed. For each type of

```
/* divide the work into chunks */
int chunk_size = size / num_kernels;

/* create CUDA streams */
cudaStream_t *stream = (cudaStream_t *) malloc(num_kernels
                        * sizeof (cudaStream_t));

for (i = 0; i < num_kernels; i++)
  cudaStreamCreate(&stream[i]);

/* launch the asynchronous memory copies and map kernels */
for (i = 0; i < num_kernels; i++)
  cudaMemcpyAsync(array_d + i * chunk_size,
                  array_h + i * chunk_size,
                  sizeof (float) * chunk_size,
                  cudaMemcpyHostToDevice,
                  stream[i]);

for (i = 0; i < num_kernels; i++)
  map_kernel<<<grid, block, 0, stream[i]>>>
            (array_d_map, /* remapping destination    */
             array_d,     /* input array              */
             i,           /* chunk index              */
             chunk_size,  /* chunk size               */
             num_kernels  /* num simultaneous kernels */);
```

Figure 4: CUDA streams are utilized to overlap chunk transfer with remapping. This example assumes that the entire work size can be evenly divided by the number of chunks

layout transformation, we provide a corresponding index transform function to achieve this functionality. For example, indexing a specific data element, `array[index]`, is achieved after the layout transformation with `array[index_transform(index)]`. This is the only change required to apply to the original GPU kernel code.

Figure 5 shows an example of the necessary changes to the *k-means* distance kernel and its associated index transformation function (a row-major to column-major transformation). We apply the layout transformation on the `feature` array, which is the primary data storage structure in this implementation. To access an element, the general form `feature_remap[transform_row2col (index, npoints, nfeatures)]` is used in place of the basic `feature[index]` lookup. Programmers can choose to manually modify the index without using a Dymaxion index transformation function (e.g. swapping the loop index in the manual code example; see Figure 5), only if they know exactly how a specific layout is optimized by the remapping function on a particular platform (e.g., DRAM parallelism and memory alignment). Because different platforms may prefer different layouts, the index transform function is preferable, as it maintains code portability across platforms without any need of manual effort. It is also a convenient tool to help programmers transform complicated index term and will be needed when implementation details are hidden from programmers.

3.3 Dymaxion API Design

Dymaxion currently optimizes single-dimension linear memory accesses (e.g. `array[index]`) to GPU global memory for various access patterns. The framework can be extended to support other memories, for example texture memory. There are several important design goals we used to guide our API development:

- Dymaxion should provide abstractions for specifying various access patterns, and the implementation of Dymaxion should rely on and can be optimized for different architectural details.
- Programmers should not be required to program with Dymaxion, which is primarily for optimization.
- The implementation should *be an API*, with a small set of extensions to existing languages, not an entirely new language.

Map Functions:

```
void map_row2col(  void       *dst,
                   const void *src,
                   unsigned   height,
                   unsigned   width
                   type_t     type);

void map_diagnal(  void       *dst,
                   const void *src,
                   unsigned   dim,
                   type_t     type);

void map_indirect( void       *dst,
                   const void *src,
                   const void *index,
                   unsigned   size,
                   type_t     type);

void map_arrstruct(void       *dst,
                   const void *src,
                   unsigned   argc,
                   arg_list   *list);
```

Index Transform Functions:

```
unsigned transform_row2rcol(unsigned index,
                            unsigned height,
                            unsigned width,
                            type_t   type);

unsigned transform_diagonal(unsigned index
                            unsigned height,
                            unsigned width,
                            type_t   type);

void     *transform_struct( void     *array,
                            unsigned tid,
                            unsigned num_mem,
                            unsigned num_nodes,
                            unsigned mem_offset,
                            type_t   type);
```

Dymaxion is not restricted to GPU use, but also applicable to other heterogeneous platforms. Compared with compiler-based tools, Dymaxion gives programmers more control while saving them significant optimization effort. The Dymaxion framework consists of two major parts: 1. A set of remapping functions to direct data remappings, and 2. associated index transformation functions.

The function list shows the current API functions implemented in Dymaxion. So far we have implemented our API for *row-major order to column-major order*, *diagonal-strip*, *indirect*, and *array-of-struct* transformations, which cover the access patterns common in many scientific applications. Note that API functions such as `cudaMemcpy()` in CUDA and `clEnqueueMapBuffer()` in OpenCL are special cases of memory mappings which map a linear region of memory space in the host to a region on the device. We do not think this is an complete list of API functions; Dymaxion is extensible to other access patterns (e.g. Morton and other space-filling curves, graph traversal), a task we leave for future work.

At the same time, there is a way in Dyamxion for programmers to define their own memory mapping that is not supplied by the API. For instance, programmers can write a GPU remapping function (equivalent to *map_kernel* in Figure 4), following the declaration rules of user-defined function in Dymaxion. As shown in the following example, the function pointer to this GPU remapping function will be used to pass to a Dymaxion `map_user` function, which automatically handles the overlapping of memory transfer and remapping.

```
void     map_user(void         *dst,
                  void         *src,
                  unsigned int height,
                  unsigned int width,
                  type_t       type,
                  usr_func_ptr map_kernel)
```

Original Version

```
__global__ kmeans_distance(float *feature_d, ...){
  //feature_d is the original array
  int tid = BLOCK_SIZE * blockIdx.x + threadIdx.x;
  /* ... */
  for (int l = 0; l < nclusters; l++) {
    index = tid * nfeatures + l;
    ...feature[index]...
  }
}
```

Dymaxion Version

```
__global__ kmeans_distance(float *feature_remap, ...){
  //feature_remap is the remapped array
  int tid = BLOCK_SIZE * blockIdx.x + threadIdx.x;
  /* ... */
  for (int l = 0; l < nclusters; l++) {
    index = tid * nfeatures + l;
    ...feature_remap[transform_row2col(index,
                                       npoints,
                                       nfeatures)]...
  }
}
```

Manually Mapped Version

```
__global__ kmeans_distance(float *feature_remap, ...){
  //feature_remap is the remapped array
  int tid = BLOCK_SIZE * blockIdx.x + threadIdx.x;
  /* ... */
  for (int l = 0; l < nclusters; l++) {
     index = l * npoints + tid;
    ...feature_remap[index]...
  }
}
```

Figure 5: The GPU kernel code examples for original, Dymaxion and manually-mapped versions

In this paper, our study is restricted to loop-based algorithms, which possess a single major access pattern that dominates computation and thus typically requires only one copy for each remapped data structure. A challenge arises when applications present multiple access patterns in accessing a single data structure. Sometimes it is beneficial to keep separate mappings for each pattern, and special care is needed to maintain consistency among the copies. But even in the presence of multiple access patterns, one desirable layout for the "major" access pattern may still generate better overall performance. Determining the performance benefits of one or many remappings depends on the degree of reuse of different access patterns in each particular application. We leave this for future work.

4. EXPERIMENT SETUP

Our results are based on execution on NVIDIA GeForce GTX 285 and 480 GPUs. The 285 has 240 cores with a 1.48 GHz shader clock, 16 kB shared memory and 1 GB device memory. The 480 has 480 cores, a 1.4 GHz shader clock, 64 kB configurable on-chip cache (shared memory + hardware cache), 768 kB shared L2 cache and 1.6 GB device memory. We use CUDA 3.1 and GCC 4.2.4 with the -O3 flag to compile our programs. To demonstrate the benefits of our framework, we also report the number of global memory loads and stores before and after using Dymaxion. This is measured by using the CUDA profiler with `CUDA_PROFILE=1` on the 285. The CPU we use is an Intel Core2 Quad CPU with a clock of 2.66GHz and a 3MB L2 cache. The results are timed on the main computational loops of the applications and include PCI-E transfer and GPU kernel execution. Also, this study is restricted to cases in which the combined memory spaces consumed by an applications working set and memory remapping does not surpass the capacity of GPU device memory.

5. ACCESS PATTERNS AND EXPERIMENTAL RESULTS

In this section, we report the performance improvements of Dymaxion for different memory patterns, each with a widely used, representative application.

5.1 Row-Major Order to Column-Major Order Remapping

Figure 6 shows a conceptual view of a row-major to column-major transformation. Essentially, a 2-D array with column-major order is a 90-degree transpose of a row-major version of the same data. From an algorithmic perspective, programmers see no difference in the two layouts (assuming of course that the program accesses the data to match the layout!); however, for regular row-wise or column-wise accesses, these two organizations are crucial to memory locality and performance. Switching from row-major order to column major order, the relationship between the new and old array index is described with

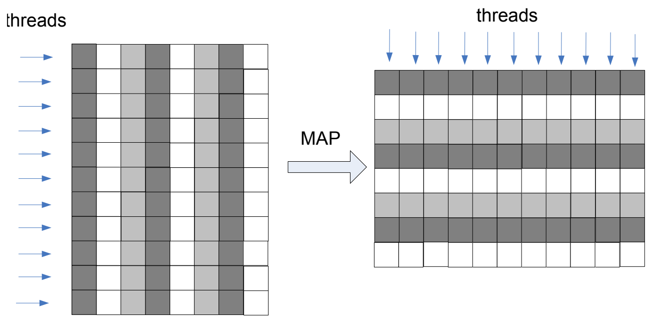

Figure 6: Row-major to column-major transformation.

```
new_index = height * (old_index % width) +
                     (old_index / width)
```

where all operations are integer and implemented by the index transform function `transform_row2col()` in the API list.

5.1.1 K-means

In the single-device Rodinia [3] GPU implementation, data are partitioned according to thread blocks, with each thread associated with one data element. The task of searching for the nearest centroid to a given element is independent of all others. We discuss the k-means implementation in detail in an earlier work [3, 4].

Programmers often prefer to store data in a 2-D arrays with each row representing a data object and each column representing a feature. Such a layout tends to be inefficient on the GPU; for instance, when threads calculate the distance of individual elements to centroids, they access whole array rows, which are often spread among multiple memory transactions. This is shown on the left in Figure 6. On the other hand, inter-thread locality is improved through coalescing after remapping the array into column-major order, shown on the right in Figure 6. This example presents a mismatch between the data affinity relationships inherent in the algorithm and the locality characteristics imposed by the mapping of SIMD operations to the DRAM organization.

We applied Dymaxion to the naïve k-*means* GPU implementation from Rodinia. Figure 7 shows the performance we obtained for the naïve implementation and the one using Dymaxion. This figure also gives the breakdown of execution time in terms of PCI-E transfer, remapping, and computation. We vary input sizes from 64 k to 256 k elements. In our experiments, the performance of the new version always outperformed the original implementation. On the GTX 480, the performance of the GPU kernel improves by an average of 3.11× due to better coalesced memory accesses. Considering layout remapping and PCI-E overheads, the overall performance improves an average of 30.6%. The combined PCI-E transfer plus layout transformation incurs only an average of 5.8% overhead when compared with the PCI-E transfer of the original implementation.

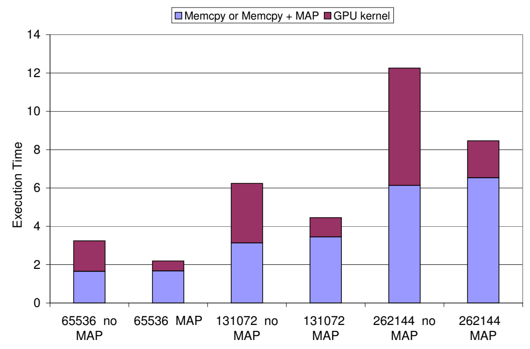

Figure 7: The y-axis represents the execution time of one iteration of the k-*means* distance calculation. Execution time is measured for both the original implementation and the port to Dymaxion. *Memcpy+MAP* represents the total amount of time due to layout remapping and data transfer. Because these two operations overlap and CUDA only provides timing for the completion of a whole stream, we measure the end-to-end time and compare it against the original data transfer (i.e. *Memcpy*)

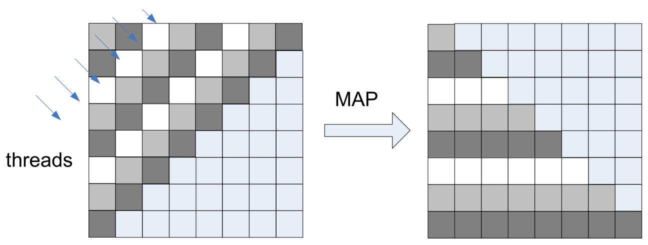

Figure 8: Diagonal strip matrix transposition.

5.2 Diagonal-Strip Remapping

Often in dense linear algebra and in dynamic programming algorithms, loops manifest memory access patterns other than regular row- or column-wise traversals; however, their access patterns *are* well defined. For example, some applications traverse arrays with constant, non-unity strides. One example is a diagonal strip traversal, which is the result of a constant stride with size $(columns-1)$. Such patterns tend to have very poor data locality. A diagonal strip is a special case of a strided access.

5.2.1 Needleman-Wunsch

Needleman-Wunsch is a global optimization method for DNA sequence alignment. Potential sequence pairs are organized in a 2-D matrix. The algorithm has two major phases: 1. the algorithm fills the matrix with scores in parallel, which represent the value of the maximum weighted path ending at that cell; and 2. a traceback process is used to find the optimal alignment for the given sequences [4]. Our implementation [4] takes advantage of the GPU's on-chip shared memory to improve program locality and reduce memory latencies; block-level parallelism within *Needleman-Wunsch* is also exploited. In this paper, we focus on optimizing *Needleman-Wunsch* through efficient memory coalescing starting from a Rodinia version with only global memory accesses.

Figure 8 illustrates the memory access patterns of *Needleman Wunsch*. This figure shows the upper-left triangular region of the 2-D matrix and its associated transformation under Dymaxion. For this particular access pattern, the relationship between the new and old array index can be described by the equation

```
new_index = dim * ((old_index % dim)  +
                   (old_index / dim)) +
                    old_index / dim
```

This transformation is achieved via the `transform_diagonal()` function in the API list. Prior to the layout transformation, parallelism exists within each diagnal strip, and each thread is assigned to compute one data element. Using our API function, programmers can make a 45 degree transposition of the matrix. The resulting layout allows threads to concurrently access data elements within the same row. In *Needleman-Wunsch*, the result of the GPU computation must be copied back to the CPU for the serial trace back; therefore, at the end of the GPU kernel, a reverse transposition is applied.

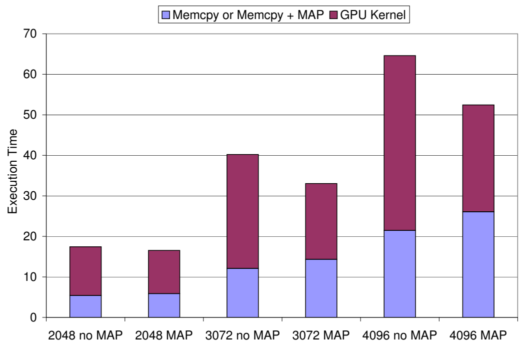

Figure 9: The y-axis represents the execution time of the *Needleman-Wunsch* kernel. Execution time is measured for both the original and the Dymaxion implementations.

We applied Dymaxion on the original, naïve *Needleman-Wunsch* GPU implementation. Figure 9 shows a performance comparison between the original implementation and the one using Dymaxion. We varied the input sizes from 2048^2 to 4096^2 data elements. On the GTX 480, the kernel performance improves by an average of 42.2%. The overall improvement averages 14.0% after accounting for PCI-E transfer and layout reorganization. The combined PCI-E transfer plus layout transformation incurs an average of 16.1% overhead when compared with the PCI-E transfer of the original implementation. Also, the best-performing *Needleman-Wunsch* version in Rodinia is 25% faster than the current Dymaxion version, because it uses the GPU shared memory, which Dymaxion does not support now, and which we leave for future work.

5.3 Indirect Remapping

Scatter and gather are two fundamental operations in many scientific and enterprise computing applications. They are very common in sparse matrix, sorting and hashing algorithms [8]. Accessing randomly-distributed memory locations makes poor use of GPU memory bandwidth. We evaluate a sparse matrix-vector multiplication (*SpMV*) to demonstrate Dymaxion's support for gather operations. A similar approach can be applied to scatter operations as well.

5.3.1 Sparse Matrix-Vector Multiplication

Our implementation adopts the compressed row format (*CSR*) to represent the sparse matrix [2, 8]. A 2-D sparse matrix, `M`, is encoded using three arrays: `DATA`, `ROWS`, and `COLUMN`. The nonzero elements of `M` are stored in the compressed data array `DATA`. Data in `ROWS[i]` indicate where the `i`th row begins in `DATA`. `COLUMN[i]` indicates the column of `M` from which the element stored in `DATA[i]` comes [22].

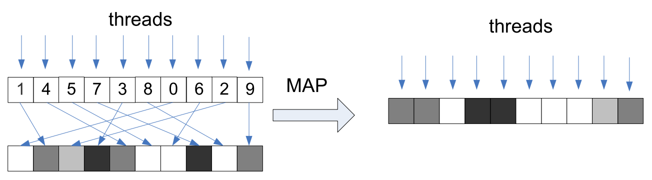

Figure 10: Indirect remapping for gather.

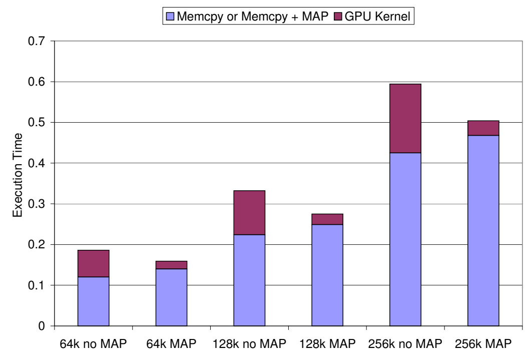

Figure 11: The y-axis represents the execution time of *SpMV*, not including reduction.

We use a similar algorithm to those described in previous work [8, 21], which computes the multiplication `W = M * V`. The algorithm computes `W` in two steps: 1. compute the partial multiplication results and store them in array `R` where `R[i] = (DATA[i] * V[COLUMN[i]])`; 2. perform a reduction stage on the partial results in `R` [8]. In the first stage, `V` is first transfered to the GPU and then we perform the indirect transformation on `COLUMN`, which is chunked and transfered to the GPU, overlapping with a gather from `V` to `V'`; this stage is handled by the `map_indirect()` function. On the kernel side, after remapping, users can directly access `V'[i]` with continuously gathered data, instead of `V[COLUMN[i]]`.

Figure 11 shows the performance improvements and execution time breakdowns for the sparse matrix-vector multiply. We varied the input sizes from 64 k to 256 k data elements. Again, the performance of the implementation with Dymaxion outperforms the original implementation for all inputs. The new GPU kernel, benefiting from coalesced memory accesses, improves $4.1\times$ from the original GPU kernel on the GTX 480. The overall performance, including the PCI transfer and layout remapping, improves by an average of 15.6%. The combined PCI-E transfer plus layout transformation incurs an average of 10.2% overhead when compared with the PCI-E transfer of the original implementation.

5.4 Struct-Array Transformation

```
#define NUM_ELEM 256
struct my_struct_t {
  float a;
  float b;
  int   c;
  int   d;
} mystruct[NUM_ELEM];
```

```
#define NUM_ELEM 256
struct my_struct_t {
  float a[NUM_ELEM];
  float b[NUM_ELEM];
  int c[NUM_ELEM];
  int d[NUM_ELEM];
} my_struct;
```

A record or structure (`struct` in C) is an aggregate type which can store multiple data members grouped together under one name. The code on the left above shows an array of structures of length `NUM_ELEM`, each of which contains two floating point and two integer members. In algorithms where the elements are independent, each can be assigned individually to a thread for computation. But because the structure members were laid out contiguously, multiple thread accesses to the same member of different structs may exhibit poor data locality.

Organizing data as a structure of arrays is often preferable to an array of structures for memory access efficiency. We provide a simple API which facilitates this transformation for GPU computation. Currently, Dymaxion only supports structures that contain non-aggregate data members, because C provides limited capability to determine the type of variables at runtime. We created an enumerated type which numbers various commonly used built-in types. Users are asked to provide structure details by passing a list of members and their types to the API function. The transformation is achieved by moving data from the array of structures to a single linear memory region, saving all the raw data. For the GPU kernel, we provide index transform functions to access data elements with information such as number of nodes and member offset. The kernel returns a pointer to the location of the resultant value.

5.4.1 Nearest Neighbor

Nearest neighbor (*NN*) is an algorithm to find the k-nearest neighbors of a data element in an abstract space. Our parallel NN implementation has two major phases: 1. the parallel phase calculates the Euclidean distances from all the data to specified data; and 2. the reduction phase sorts data in order of ascending distance. We are interested in the first phase. Its distance calculation is similar to that of k-means, differing primarily in representation, as NN uses an array of structures.

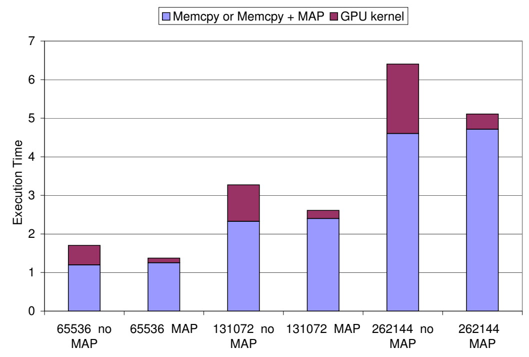

Figure 12: The y-axis represents the execution time of the first phase of the *NN* implementation.

Figure 12 shows the performance we obtained for the naïve implementation and the one with reorganized structures. The input sizes are varied from 64 k to 256 k elements. For all inputs, the performance of the optimized version outperforms the original implementation. On the GTX 480, the GPU kernel was able to achieve $4.4\times$ speedup over the original version, and the performance, including remapping and the data transfer, increases by 20%. The combined PCI-E transfer plus layout transformation incurs only an average of 3.4% overhead when compared with the PCI-E transfer of the original implementation.

5.5 The Benefits of Memory Remapping

To further evaluate the benefits of Dymaxion, we also perform the same set of experiments on an NVIDIA GTX 285 GPU. Figure 13 shows the speedups of the GPU kernels with Dymaxion against their original implementations for our applications on both the GTX 480 and GTX 285. The speedups range from $1.4\times$ to $4.4\times$ on the 480 and from $2.1\times$ to $3.0\times$ on the 285. The performance benefits are due to a better match between the memory access patterns and layouts of data structures after applying Dy-

Table 2: Total number of loads and stores of different application-input pairs reported by the CUDA profiler

Application	Input Size	Without Remapping Opt.	With Remapping Opt.
k-means	64 k	289600	68224
	128 k	590784	133824
	256 k	1193152	267648
NW	2048	1536	325
	3072	2432	497
	4096	3200	625
SpMV	64 k	9042	2496
	128 k	19084	4896
	256 k	35846	9792
NN	64 k	188032	60800
	128 k	376064	124132
	256 k	744896	250496

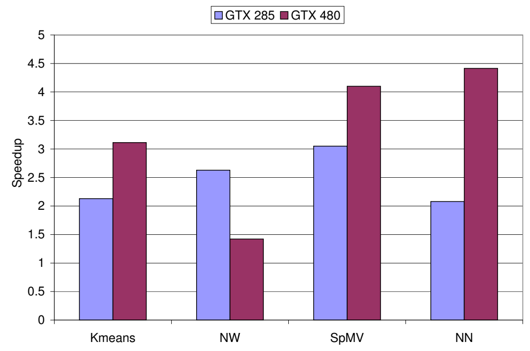

Figure 13: The y-axis represents the speedup of the GPU kernels with Dymaxion against the original implementation.

maxion, which leads to improved memory coalescing in CUDA. We also use NVIDIA's CUDA profiler to characterize the GPU kernels. As shown in Table 2, the number of global memory loads and stores is reduced significantly using Dymaxion, with approximate reductions of $4.2\times$ for k-means and $3.6\times$ for SpMV.

5.6 Chunking Overhead and Parameters for Remapping

Using a microbenchmark, we measured the overhead of conducting PCI-E transfers by breaking up the data into chunks. We consider two scenarios: one in which chunks are transferred with synchronous `cudaMemcpy()` versus another combining streaming and `cudaMemcpyAsync()`. We repeatedly iterate over the loop, transferring one chunk per iteration. Figure 14 shows the normalized throughputs measured while transferring a total of 16 MB of contiguous data. We varied the number of chunks from 1 to 512. Conducting smaller data transfers incurs more performance overhead, and the throughputs of both scenarios begins to degrade significantly at about 16 chunks. The latter scenario, streaming + `cudaMemcopyAsync()`, achieves better throughput with an average of 16.7% improvement over the case using synchronous transfers. The benefit is due to multiple streams of chunks and reduced overhead due to the queuing of asynchronous memory calls.

We also investigate an approach to optimize the performance of remapping, which maps the locations of data elements from one memory space to another. In the GPU implementation of Dymaxion, the remapping is achieved through a call to a GPU kernel. Though these kernels are simple, the performance can vary considerably depending on the amount of work done per thread. Merrill et al. [15] use a set of techniques to optimize GPU-to-GPU data movements. Figure 15 shows the normalized performance of a row-major order to column-major order remapping as a function

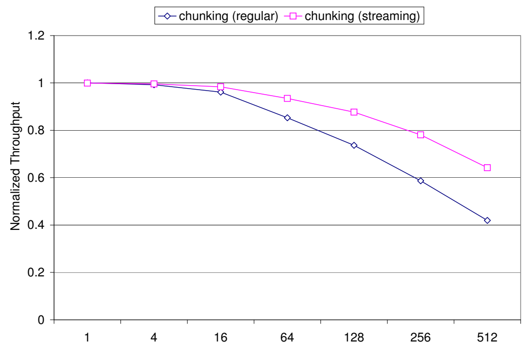

Figure 14: The y-axis represents the normalized throughput comparing chunking with synchronous memory transfers and chunking with asynchronous memory transfers (streaming)

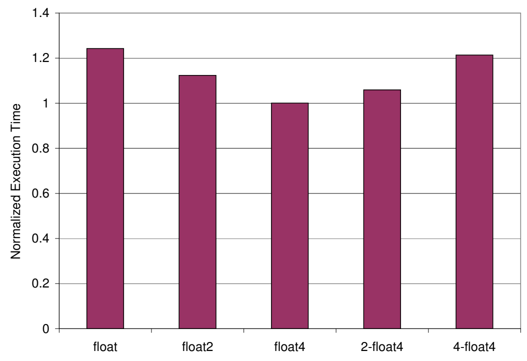

Figure 15: The vertical axis represents the performance of remapping (row2col) as a function of the number of bytes (4–64 B) each thread reads and writes. The execution times are normalized to the best performance point. The input is a 16 k$\times$16 float matrix.

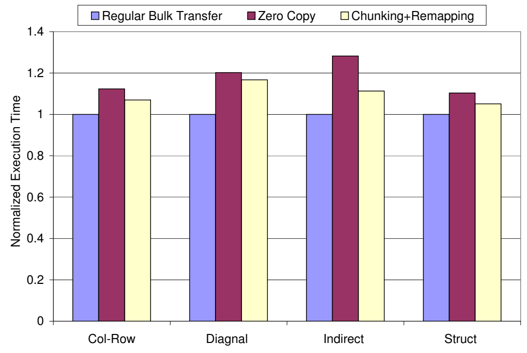

Figure 16: The y-axis represents the normalized execution time of three types of data transfers. The baseline is the time consumed by the bulk data transfer in the original implementation

of the number of bytes (4–64 B) each thread reads and writes with a thread-block size of 256. The performance differences between best and worst performing points can be as large as 24%.

5.7 Gathering Data through Zero-Copy

In Section 3.1, we discussed the approach of hiding remapping latency by overlapping PCI-E data transfer and remapping each chunk, which can be applied in general circumstances. In this section, we use a CUDA-specific feature, *zero copy*, to achieve the same goal of gathering data into contiguous segments. *Zero copy* allows GPU threads to directly access host memory which are page-locked [6]. In the former approach, we transfer each chunk from system memory to an intermediate staging buffer on the GPU and subsequently perform a remapping by saving the final remapped data to a destination buffer.

Using *zero copy*, we launch the remapping kernel with threads sourcing data directly from system memory and then storing the data into the destination buffer in GPU device memory. This saves the memory duplication overhead on the GPU and also obviates the need for each data element to be read and written twice; however, the drawback is that many smaller transactions are needed. Figure 16 shows normalized performance results comparing three types of data transfers. *Zero copy* gathers data from non-contiguous memory regions and incurs an average of 7% performance degradation when compared with chunking + remapping; however, all of our applications still get an average speedup of 16.2% on the overall performance due to improved GPU kernel execution time. In other words, our proposed remapping approach has an advantage–despite the extra copy step–because the GPU, working from GPU memory, can achieve higher throughput on data layout transformation than attempting to perform transformation as part of the PCI-E transfer. This is thanks to the GPU's higher memory bandwidth and parallelism, coupled with the chunking approach's ability to hide this latency.

6. CASE STUDY: CPU-GPU SIMULTANEOUS EXECUTION

In the previous section, we show the performance improvements of offloading work to the GPU and applying our Dymaxion framework. In this section, we present a case study using k-means, spreading the workload simultaneously across the CPU and the GPU, which is desirable for two reasons: some CPUs are capable enough to contribute meaningfully to overall performance, and using the CPU will reduce the amount of data that needs to be transferred to the GPU. Some programming models, such as OpenCL [18], support heterogeneous systems, allowing programmers to write one piece of code that is portable to multiple platforms. Unfortunately, one implementation of the compute kernel is usually developed assuming a single data layout, suggesting that it would not work well across diverse platforms.

The Dymaxion framework is useful in this regard, maintaining data coherence while optimizing access patterns across the CPU and GPU. For instance, in a multithreaded k-means CPU implementation, each CPU thread is responsible for processing one region of data. Each thread processes one data element and proceeds to process the next element and so on within its own region. Therefore, on the CPU, k-means favors a row-major array organization, and the features of a single data element can reside contiguously in cache lines to generate better data locality for distance calculations. This is quite different from the GPU's preference, as discussed previously, for a column-major ordering. Our tests show that a column-major layout degrades CPU performance approximately 2× compared with row-major order, while a row-major layout degrades GPU performance approximately 50% compared with column-major order.

Previous work, including Qilin [14] and Merge [12], presents work-spreading across the CPU and the GPU, but we are unaware of any previous work evaluating the performance impact of different memory mappings when concurrently scheduling workloads on heterogeneous compute resources.

Figure 17 illustrates this concept with the k-means implementation. To ease the computational domain partitioning over multiple

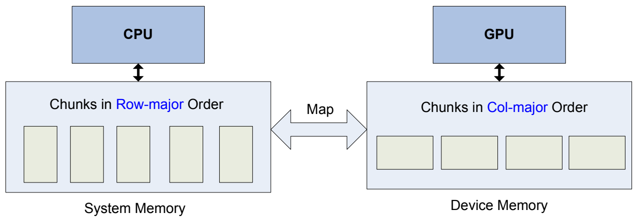

Figure 17: The CPU and the GPU prefer different mappings. We divide the data structure into smaller chunks and schedule them onto the CPU and the GPU

Table 3: Workload Ratios of K-*means*

Core Combination	CPU	GPU
CPU + GPU (Row-major Order)	35%	65%
CPU + GPU (Column-major Order)	25%	75%

devices, we divide the main data structure into smaller chunks and schedule them on different devices. Load balancing across devices is an interesting research issue in itself and is not the focus of this paper. In our experiment, the baseline is a CPU implementation whose data structure is stored in a row-major order; for each chunk dispatched onto the GPU, Dymaxion is applied to remap the chunk into column-major order for efficient GPU execution.

Figure 18 shows the normalized execution time for the k-means distance kernel with 1.25 M data points and 16 features. When both the CPU and GPU use row-major order, the simultaneous CPU-GPU execution improves the performance by 20% over GPU-only execution. After applying Dymaxion to obtain column-major layout, the GPU-only execution obtains 15% performance improvement over simultaneous CPU-GPU execution with row-major-only order layout. If the CPU uses the row-major layout and GPU uses column-major layout, scheduling k-means on the CPU and the GPU further improves the performance by 18% over the GPU-only, column-major layout. As shown in Table 3, for the CPU + GPU (row-major order) configuration, the portions of the workloads mapped to the CPU and GPU are 35% and 65%, respectively. Switching to the CPU (row-major order) + GPU (column-major order) configuration, the portions of the workloads mapped to the CPU and GPU change to 25% and 75%.

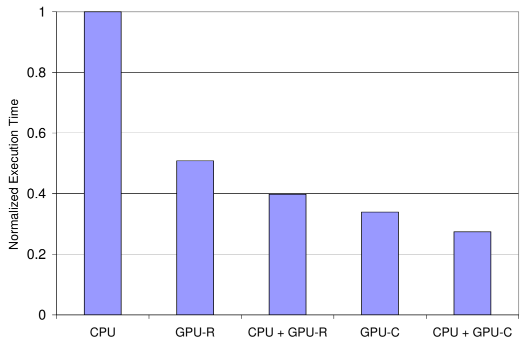

Figure 18: The normalized execution time of the CPU-GPU simultaneous execution for one iteration of k-means. In all the cases, the CPU uses row-major order. For the GPU implementation, we use two layouts: *R* represents row-major order while *C* represents column-major order

7. DEVELOPMENT COST

The goal of Dymaxion development is to improve the productivity of programmers in optimizing memory accesses. Programming effort must be taken into account in the evaluation of our API's utility. Because it is difficult to get accurate development-time statistics for coding applications, we use *Lines-of-Code* (LOC) as our metric to estimate programming effort. Table 4 shows the number of changed lines of code for the four applications used in this study. For all of the applications, Dymaxion required only 6-20 lines of changes. This suggests the programmer effort of applying our API is trivial compared with the performance gains.

Applications	Kmeans	NW	NN	SpMV
LOC	7	18	20	6

Table 4: Development cost measured by number of modified lines of code.

8. RELATED WORK

Previous work investigates how to optimize data organization for efficient memory accesses. An early report by Leung and Zahorjan [10] discusses how array elements should be laid out in memory to improve spatial locality for accesses in nested loops. Impulse [22] proposes application-specific optimizations through configurable physical address remapping by supporting prefetching at the memory controller. Sung et al. [20] investigated a compiler approach for layout transformation for GPU kernels, focusing on structured-grid applications.

Jang et al. [9] use a mathematical model and associative algorithms to analyze data access patterns and target loop vectorization and GPU memory selection with different patterns. The linear, shifted, and strided access patterns [9] can be handled by either the row-major or column-major mapping alone in our framework. Zhang et al. [21] proposes a dynamic approach to reduce irregularities in GPU programs. Latency hiding is achieved by overlapping kernel computation and memory transfer, and requires splitting the kernel, difficult for certain applications with dependencies. These approaches maintain many duplicate data copies for fine-grained data reordering. In contrast to these projects, we optimize memory efficiency by allowing programs to provide hints about memory access patterns. Our framework is based on a set of commonly used data layouts and access patterns in scientific applications. We propose that memory remapping and related latency-hiding techniques be implemented during CPU-GPU data communication. Our technique is more general and does not break the integrity of compute kernels, and the memory overhead is small. Also, none of the previous work evaluates the memory mapping issue when scheduling and balancing workloads on both the CPU and the GPU.

Other APIs for GPU computing have been proposed. Thrust [11] is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library and based on vector data structures. Sengupta et al. [19] implemented the classic scan operation using CUDA, providing a set of library functions to deal with applications with more irregular data structures. There are also several libraries for FFT and BLAS operations and for video codecs [17]. These APIs offer abstractions for either data structures or domain operations. Though the ways memory accesses are handled are transparent to users of these frameworks, they are usually optimized for specific layouts, for example column-major storage is used in CUBLAS [17]. Our approach is more general, with a focus on various memory layouts and accesses, and also more useful for bridging the gap between different devices.

9. CONCLUSIONS & FUTURE WORK

In this paper, we propose the Dymaxion framework to optimize the efficiency of DRAM accesses through memory layout remapping and index transformation. We hide the overhead of remapping through data structure chunking and by overlapping with the CPU-GPU PCI-E data transfer. Usage of our API requires only minimal changes to the original implementations. The four applications we evaluate, each with a unique access pattern, achieve an average of $3.3\times$ speedup on the compute kernels and 20% overall performance improvement, including the PCI-E transfer, on an NVIDIA GTX 480 GPU when compared with their original implementations. The overall benefit is limited by PCI-E overhead, so the benefit will improve as the PCI-E protocol improves or the GPU becomes a peer with the CPU. Also, Dymaxion is a convenient building block to ensure data coherence between the CPU and the GPU for heterogeneous computing; a remapping is needed when writing data to the GPU while a reverse-remapping is needed when reading data from the GPU. We plan to extend Dymaxion to support the transformation of multidimensional arrays and special memories such as texture and shared memory. Dymaxion will be released on line at http://lava.cs.virginia.edu/dymaxion.

Today's GPU programming models require programmers to manually optimize memory access patterns. Commercial GPU compilers do not yet support Dymaxion-like memory-remappings. We anticipate that the techniques used in this paper can be further integrated into compiler frameworks for automated memory remapping. For instance, an OpenMP-like directive can be used to specify the preferred data structure organization for the CPU or the GPU. The compiler can then automatically insert the Dymaxion-like remapping and transformation.

There are also several other directions of future work we plan to explore. This paper focuses on a single machine node. Although MPI can launch the same CUDA operations (including Dymaxion remappings) on each node in a cluster, a global, cross-cluster approach may allow further optimizations, especially when cross-node data transfers and system-level interconnect are considered. Additionally, the dynamic detection of application access patterns is a very promising research direction. Currently, Dymaxion requires programmers to manually choose the appropriate API calls for data rearrangement; fortunately, Dyamxion allows programmers to easily roll back to the original version whenever the performance is not satisfactory. Also, because remapping of very large data arrays may introduce additional power overhead, future work will explore the energy efficiency. We also wish to explore opportunities for remapping among different levels of the memory hierarchy, especially when heterogeneous processors share memory (e.g. in AMD Fusion [1]) or even a last-level cache.

Acknowledgments

This work was supported in part by NSF grant CNS-0916908 and by a grant from NEC Research Labs America. We would also like to thank the reviewers for their constructive comments.

10. REFERENCES

[1] AMD Fusion APU. Web resource. `fusion.amd.com/`.

[2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec 2008.

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

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors using CUDA. *J. Parallel and Dist. Comp.*, 68(10):1370–1380, 2008.

[5] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS*, April 2011.

[6] CUDA C Programming Best Practices Guide. Web resource. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf`.

[7] NVIDIA CUDA Programming Guide. Web resource. `http://developer.nvidia.com/object/gpucomputing.html`.

[8] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *SC*, Nov 2007.

[9] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data parallel architectures. *TPDS*, 22:105–118, 2010.

[10] S. T. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR 95-09-01, University of Washington, Sept 1995.

[11] The Thrust library. Web resource. `http://code.google.com/p/thrust/`.

[12] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A programming model for heterogeneous multi-core systems. In *ASPLOS*, Mar 2008.

[13] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[14] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO-42*, 2009.

[15] D. Merrill and A. Grimshaw. Parallel scan for stream architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, Dec 2009.

[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[17] NVIDIA CUDA. Web resource. `http://www.nvidia.com/object/cuda_home_new.html`.

[18] OpenCL. Web resource. `http://www.khronos.org/opencl/`.

[19] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *GH*, Aug 2007.

[20] I-J Sung, J. A. Stratton, and W-M W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *PACT*, Sept 2010.

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

[22] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The impulse memory controller. *IEEE Trans. Comp.*, 50(11):1117–1132, 2001.