Dymaxion++: A Directive-based API to Optimize Data Layout and Memory Mapping for Heterogeneous Systems

Shuai Che[†], Jiayuan Meng[*] and Kevin Skadron[⋆]
Shuai.Che@amd.com, jmeng@alcf.anl.gov and skadron@cs.virginia.edu

AMD Research[†], Argonne National Laboratory[*] and Dept. of Computer Science, University of Virginia[⋆]

***Abstract*—There has been a growing trend in using heterogeneous systems with CPUs and GPUs to solve diverse compute problems. However, high application performance on these platforms relies on efficient memory accesses. For many applications, CPUs and GPUs prefer different memory mappings and data-structure layouts. This in turn requires developers to use device-specific strategies for memory access optimizations. Achieving both code and performance portability becomes a challenge for heterogeneous computing.**

This paper proposes a directive-based API, *Dymaxion++*, which enables programmers to optimize memory access patterns across devices with a simple interface. Use of *Dymaxion++* requires only minimal modifications to existing codes with a small set of pragma extensions. The current framework augments the original *Dymaxion* framework [6] with a clean abstraction backed by a source-to-source code translator. *Dymaxion++* also provides additional programming features to map data structures to GPU's hybrid memory spaces (e.g. texture and constant memory) for different uses. Additionally, data layout transformation is enabled while exchanging data between GPU scratchpad and device memory as well as between system memory and device memory.

I. Introduction

Developing codes that work efficiently across different devices is challenging, especially for heterogeneous systems with diverse accelerators (e.g., AMD APU [1], IBM Cell [11], and Intel MIC [9]). In particular, the fast growing core counts in today's multicore and manycore architectures have exposed significant pressure on the memory subsystem. Memory bandwidth and latency are improving at a relatively slower pace, which limits the overall system throughput. High application performance relies heavily on efficient memory bandwidth utilizations. Though GPUs usually have a wider memory interface than CPUs, performance would be suboptimal in the presence of insufficient memory coalescing [10], [20], [23].

To achieve good performance, different architectures may prefer distinct access patterns and data structure layouts. This requires device-specific optimizations for memory references. In addition, GPUs' hybrid memories (e.g., scratchpad, texture and constant memories) present specialized access patterns. However, codes optimized for these access patterns may not perform well on another device [6]. Also, programming APIs are designed to program different devices but do not solve the issue of performance portability. For instance, OpenCL has been released to target diverse compute devices. However, a single OpenCL implementation (e.g. with the same data-structure layout, block size, compiler options, etc.) usually does not perform equally well across devices. In particular, this becomes critical when CPUs and GPUs are coordinated on the same task. Compiler and runtime support is required to bridge the gap for heterogeneous data layouts and access patterns favored by different devices. An important optimization technique is memory remapping [6]. The technique restructures layout of data structures or mapping them to specialized spaces. However, memory remapping sometimes requires a high-level understanding of the code and can be difficult for compilers or runtime to identify automatically. The goal is to allow the programmer to specify in a generic way what they want to accomplish, e.g. a generic access pattern, and let the device specific optimizations happen transparently.

This paper addresses these concerns with a simple API, Dymaxion++, to improve memory accesses in unoptimized code. Dymaxion++ aims to achieve both good programmability and performance portability. It provides a directive-based API for programmers to express heuristics for memory remapping. In Dymaxion++, programmers are asked to employ a set of simple *pragmas* and associated *clauses* in particular regions of parallel code and direct the compiler and runtime system to make appropriate memory remapping decisions.

Dymaxion++ extends the original Dymaxion proposal [6]. However, the original version of Dymaxion requires non-trivial modifications to the source code. In contrast, Dymaxion++ allows programmers to annotate code with a much clearer interface and helps to preserve the original code structure. In addition, we include support for memory-space mapping and layout transformation across different memory spaces. The proposed programming style is general and can be integrated easily into other directive-based models (e.g., OpenMP and OpenACC) for parallel programming. Code and layout transformation is handled automatically by the framework with a source-to-source translator and a runtime system. Dymaxion++ currently targets GPUs, but can be implemented and optimized for any other device.

This work makes the following contributions:

- We present the first-step design of the Dymaxion++ API. Its high-level abstraction of memory layout and space mapping, together with the parameterized model, enable programmers to apply their domain knowledge to influence memory remapping in a flexible manner.
- We demonstrate the capabilities of Dymaxion++ as well

Part of this work was completed at the University of Virginia. The extension was done at AMD

as its code translation framework converting Dymaxion++ code into regular OpenCL/CUDA code. We present the underlying mechanism of layout reorganization for efficient inter-thread DRAM coalescing, mapping to scratchpad memory and other specialized memory spaces.
- We evaluate several compute kernels with representative access patterns and use them as preliminary case studies to present easy use of Dymaxion++. We also show their performance speedups on real hardware.

II. An Overview of Dymaxion++

Today's GPU programming models (e.g., OpenCL and CUDA) require programmers to expend significant effort to optimize memory accesses. Dymaxion++ is an extension to existing GPU programming models to allow easy optimization of memory accesses for heterogeneous systems. Applications that make use of Dymaxion++ possess a generic code form (straightforward, unoptimized version of OpenCL/CUDA implementation). Dymaxion++ also relieves programmers from dealing with error-prone and fine-grained code structuring that leads to *ad-hoc* device-specific codes.

A. Directive-based Programming

The Dymaxion++ framework consists of two major components. It features a layout transformation model and a memory-space mapping model:

- The *layout-transformation (reshape)* model allows programmers to annotate data structures to guide transformation. The data structure is reorganized into a different shape in memory that is ideal for memory accesses.
- The *memory-space mapping (place)* model allows programmers to map data structures to different memory spaces (e.g., texture and constant) without using tedious OpenCL/CUDA APIs for memory management. With this model, the program can map data structures in the device memory to specialized memory spaces for different uses, while maintaining the original code structure.

In certain scenarios, layout transformation and memory-space mapping models can be used in combination. For instance, when conducting a discrete GPU offload, beginning with PCI-E transfers to the point where data is brought to GPU cores, there can be different spaces to store remapped data structures (e.g. system, device and, scratchpad memories). Dymaxion++ directives follow the *pragma* mechanism provided by the C/C++ standards. Each directive starts with *#pragma* followed by *clauses* to specify detailed parameters. The general formats of Dymaxion++ programming model are as follows:

```
//Memory Layout Transformation Model
#pragma dympp reshape [clause]

//Memory-space Mapping Model
#pragma dympp place [clause]
```

The *reshape* construct is used to specify memory layout transformation in general. Immediately after this option, programmers need to specify the type of transformation in the clause. Programmers also are asked to provide additional information, such as the name of data structure, array dimension and data types and so on, when necessary. The current Dymaxion++ implements diverse transformations common in many scientific applications, including *row2col* (convert row-major to column-major), *diagonal* (support diagonal access in a rectangular array), *indirect* (gather data into a compact vector from non-contiguous memory locations) and *stride* (reorganize data in which adjacent accesses present a fixed stride). Our API is a high-level abstraction while the actual implementations may vary depending on specific architectures (e.g., memory alignment, address distributions, cache hierarchy, etc.). Even though the same transformation type (e.g., row2col) applies for similar devices (e.g., AMD or NVIDIA GPUs), due to their architectural differences, each device may favor device-specific parameters for layout transformation. It is possible to support different versions of underlying transformation routines for each device. These are part of the Dymaxion++ framework that are transparent to the programmer. The Dymaxion++ system chooses the appropriate version of transformation routine (e.g. row2col_amd) to generate code for the detected device.

The *place* construct is used to specify data-structure mappings to different memory spaces. The framework then conducts code translation automatically with user-provided hints (e.g., the memory space, data structure name and array dimension) to exploit GPU scratchpad, texture and constant memories. It produces code to allocate these memory buffers for the target data structure the user annotates, move data between buffers across spaces and manage memory accesses to these buffers. The space mapping is handled transparently, so programmers write generic code as usual with appropriate directives,. In addition, for the scratchpad memory, when data mapping is used together with layout transformation, two options are supported, which we will discuss in detail in Section V-A:

- Layout transformation is performed across the system and device memories by chunking data structures and overlapping them with PCI-E transfers. With the same layout, the remapped data structure is loaded from the device to scratchpad memory to take further advantage of program locality.
- Data is transferred between the CPU to the GPU with regular PCI-E transfers and layout transformation is performed when loading data from the GPU device memory to the scratchpad memory as part of the compute kernel.

In this study, as a preliminary proof of concept, our directive-based framework is built on top of existing GPU programming models, annotating OpenCL and CUDA codes. Extension to OpenMP [18] and OpenACC [16] is straightforward but beyond the scope of this paper. Pragmas are chosen for these OpenCL/CUDA annotations because OpenMP-like programming has shown convenience of pragmas for programmers. The proposed pragmas can be used in combination with existing OpenMP directives (e.g., OMP parallel for) to annotate a code block for accelerator offload. In addition,

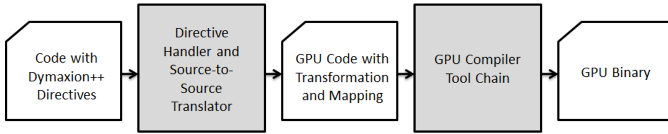

Fig. 1. A Dymaxion++ program is fed into a directive handler and source-to-source translator. It then outputs intermediate OpenCL code and further compiled to device-specific binary with a vendor's tool chain.

OpenCL is used to demonstrate code examples. They all use the *device* clause to specify the target device for which Dymaxion is being directed to optimize. It is also useful to include a "source_device" clause to specify for which the original code is written for, because OpenCL is supported by diverse devices. It can have a default device value when not used (e.g. CPU).

B. Source-to-Source Translation Framework

Figure 1 illustrates the Dymaxion++ code translation flow. A Dymaxion++ program is fed into a directive handler and the source-to-source translator that outputs intermediate OpenCL and CUDA code. It is then compiled into GPU binaries with vendors' specific tool chains.

Our source-to-source translator produces OpenCL/CUDA programs with additional code plugged in for layout transformation and memory-space mapping. For instance, when layout transformation is conducted and optimized during PCI-E data transfer, Dymaxion++ generates new code that allocates buffers on the device memory and transfers data between the host and device memory in a mapping order guided by pragma specifications. The translator inserts codes to perform memory layout transformation. The transformation is conducted with a separate kernel before the actual computation kernel. Layout transformation is achieved by making each GPU thread map (copy) one data element from the old to new memory location. During the layout transformation, we further optimize layouts with padding to meet the memory alignment requirement for better inter-thread coalescing. The code translator also generates CPU-side code, which attempts to hide the latency of data transformation by overlapping PCI-E transfers and data reorganizations [6]. Similarly, the framework can generate kernel-side code for layout transformation between the GPU device and scratch memory. In this case, an existing kernel is modified to insert transformation before the user-specified portion of the computation. For this proof of concept, rather than an integrated source-to-course compiler pass, we use *Perl* scripts to prototype simple code translations.

For the actual GPU compute kernel, another major transformation performed by Dymaxion++ is to redirect memory accesses (i.e., address redirection). Due to layout changes, array indices need to point to new locations in the remapped data structures [6]. The source-to-source translator will produce new array indices or index computations to visit the remapped data structures. Similarly, for memory-space mapping, Dymaxion++ generates necessary GPU codes to access texture and constant memories on both the host and kernel sides.

III. Experimental Setup

The experimental results are measured on real hardware with an AMD Radeon HD7950 (Tahiti) discrete GPU and a NVIDIA GeForce GTX 480 GPU. The AMD HD7950 features 28 Graphics Core Next (GCN) compute units with a total of 1,792 stream processors with an 800 MHz shader clock and 3 GB of device memory. The Geforce GTX 480 has 480 cores, a 1.4 GHz shader clock, 768 kB of shared L2 cache, and 1.6 GB of device memory. We use AMD APP SDK 2.8 with OpenCL 2.1 support and NVIDIA GPU Compute SDK with CUDA 4.0 support. In addition, this study is restricted to cases in which the memory buffer sizes consumed by applications and Dymaxion++ do not exceed the capacity of GPU memory; handling such cases is left for future work.

IV. Layout Transformation

In this section, we use several case studies to show how to leverage Dymaxion++'s directive-based interface for memory remapping. We use the same application examples as the prior Dymaxion work [6]. The codes with Dymaxion++ directives are demonstrated in OpenCL. The *reshape* construct currently supports several clauses: *row2col*, *diagonal*, *indirect* and *stride*.

A. Transformation and Latency Hiding

Layout transformation is conducted on the GPU side with built-in GPU kernel templates customizable through user-provided heuristics. Regular data transfer will be replaced by Dymaxion++ transfers and the following layout transformation on the GPU. We hide transformation latency by overlapping memory layout reorganization with chunked PCI-E transfer. This technique is studied in prior work [6], [23]. In a brief summary, when layout transformation is conducted across the system and device memory through PCI-E, we break the data into small chunks and transfer each chunk asynchronously from the CPU to the GPU one by one. Immediately after data transfer of each chunk, Dymaxion++ launches transformation kernels to reorganize data layout; each thread is responsible for relocating one data element. The transfer of the next chunk overlaps with on-going layout transformation of the most recently transferred chunk. When layout transformation happens across the GPU device and scratchpad memory, techniques such as CudaDMA can be used to achieve a similar goal, with a subset of threads dedicated to data prefetching and transformation with specific rules [3].

B. Row-major to Column-major

We use the *Kmeans* distance kernel to demonstrate the use of Dymaxion++ for *row-major-to-column-major* transformation (see Figure 2). In *Kmeans*, the task of searching the nearest centroid to a given data element is done in parallel by multiple threads [5]. The *feature* structure is stored in a 2-D array in a row-major layout with each row representing a data object and each column representing a feature. Such layout is efficient for CPU computation, because when a single thread calculates the distances of each row to the centroids, different

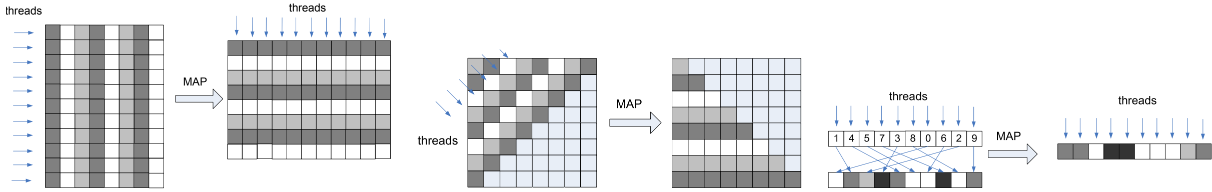

Fig. 2. Three types of Dymaxion++ transformations: *transpose*, *diagonal* and *indirect* transformations (from left to right) [6]

features of a row are laid out contiguously in a cache line for data locality. However, this is inefficient for GPU's SIMD execution; threads in a SIMD group access different rows in the array, which often are spread among multiple memory transactions. Therefore, column-major layout is desirable for *Kmeans* on the GPU.

In this case, programmers can use the *reshape* construct and *row2col* clause to optimize *Kmeans*' access patterns for the GPU. As shown in the example, the device for the target transformation is GPU. *row2col* is applied on the *feature* array, which then is reorganized into a column-major order on the GPU. The layout organization and associated latency hiding will be handled by the runtime system [6]. The source-to-source translator will generate code to translate the array indices to access the remapped *feature* array. The new index calculation follows the rule: $new_index = npts * (old_index\%nfts) + (old_index/nfts)$.

```
unsigned tid = get_global_id(0);
if (tid >= npts) return;
...
#pragma dympp device(GPU) \
        reshape row2col(feature[npts, nfts])
for (int l=0; l<nfts; l++){
  ans += (feature[tid*nfts+l]-clusters[i*nfts+l])*
         (feature[tid*nfts+l]-clusters[i*nfts+l]]);
}
...
```

Listing 1. The distance kernel of *Kmeans* (*row2col*). All the code examples in this paper are shown in OpenCL

C. Diagonal

We use the *Needleman-Wunsch* algorithm to demonstrate the use of diagonal-strip transformation in Dymaxion++ (see Figure 2). In *Needleman-Wunsch*, potential DNA sequence pairs are organized in a 2-D matrix. The first phase can be done in parallel by filling the matrix with scores, each of which represents the value of the maximum weighted path ending at each cell. *Needleman-Wunsch* traverses the computation domain in a diagonal strip manner; parallelism is present for the data elements processed within a diagonal strip. Adjacent strips must be processed serially [5].

The code example in Listing 2 shows the code region that fills the score matrix (upper-left) by accessing each element's northwest, west, and north neighbors. More efficient GPU computation is achieved when SIMD threads access contiguous data elements as well as their neighbors to maximize the benefit of DRAM coalescing. As shown in the code, Dymaxion++ provides programmers the *diagonal* clause to rotate the *score* array by 45 degrees. Similarly, our source-to-source translator modifies the original code by transparently adding extra routines for layout and array index transformation. The new index calculation for diagonal-strip transformation follows the rule: $new_index = dim * ((old_index\%dim) + (old_index/dim)) + old_index/dim$

```
unsigned int tid = get_global_id(0);
...
if (idx <= i ){
  unsigned int index = tid * dim + (i - tid);
  #pragma dympp device(GPU) \
  reshape diagonal(score[dim, dim])
  score[index]= max(score[index-1-dim]+ref[index],
                    score[index-1]    -penalty,
                    score[index-dim]  -penalty);

}
```

Listing 2. The score-filling kernel in *Needleman Wunsch* (diagonal)

D. Indirect

Gathering data from randomly distributed memory regions into a contiguous vector makes SIMD execution more efficient. The code in Listing 3 shows a simple multithreaded vector multiplication, in which indices of *v* are obtained from the *col* array and used to gather actual data. It can be used in sparse-matrix vector multiplication [6], [8], in which the algorithm computes the result in two steps, including a partial multiplication step and a reduction step summarizing the partial result.

By applying Dymaxion++'s *indirect* clause, the gathering operation (see Figure 2) can happen at the same time while conducting data communications between memory spaces. For instance, `v` is first transferred to the GPU, and then we can apply the indirect transformation on `col`, which is chunked and transferred to the GPU overlapping with a gather from `v` to `v'`. The vector `v'`, an internal structure maintained by Dymaxion++ (not shown in the code), contains continuously gathered data and is used for future accesses.

```
unsigned int tid = get_global_id(0);
#pragma dympp device(GPU) reshape indirect(v[col[dim]])
result_vector[tid] = data[tid] * v[col[tid]];
```

Listing 3. The gather kernel (indirect)

E. Other Transformations

In addition to the previous three transformations, which are useful in many applications, Dymaxion++ can be extended to support other memory remappings. This paper shows only a subset of access patterns for proof-of-concept purposes. For instance, *row2col* and *diagonal* transformations are two special cases of the *strided* access pattern (e.g. matrix[tid

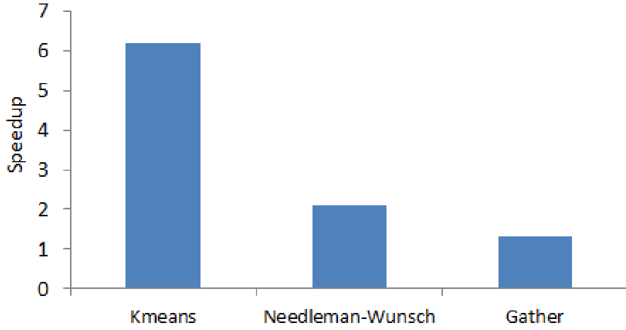

Fig. 3. Performance speedups for the GPU kernels (Dymaxion++ vs. non-Dymaxion++ versions)

* stride + j]), where *stride* represents the distance between two adjacent thread accesses. In particular, when accessing an array a[0:m*n], the stride is n for *row2col* and $(n-1)$ for *diagonal* respectively. Thus, the *reshape* construct can be extended to arbitrary strides. Ideally, the stride size after transformation should be 1 for adjacent GPU threads within a wavefront. This achieves the best memory coalescing – the goal of Dymaxion++ reorganization. In addition, our directives can be extended to support restructuring arrays of struct (AoS) to struct of arrays (SoA).

Sometimes, users prefer to define their own reorganizations to solve particular problems. The Dymaxion++ framework is extensible to user-defined transformation. In this case, users specify a *remap_func* function, which defines the relationship between the old index and the new remapped index for each data element in a linear array space.

The following code in Listing 4 shows an example of strided and user-defined transformations respectively.

```
//An example of the strided reorganization to array arr
#pragma dympp device(GPU) \
        reshape strided(arr[dim, dim, stride])

//An example of the user-defined reorganization to arr arr
#pragma dympp device(GPU) \
        reshape userdef(arr[dim, dim, remap_func(0:m*n, m,
               n)])
```

Listing 4. Strided access pattern and user-defined remapping

F. The Benefits of Layout Transformation

We applied Dymaxion++ pragmas to the naïve GPU implementations of several Rodinia compute kernels. Figure 3 shows the speedups of three GPU compute kernels with Dymaxion++ compared to their original implementations on an AMD Radeon HD7950. The speedups are 6.2×, 2.1× and 1.3× for *Kmeans*, *Needleman Wunsch* and *Gather* respectively. The performance benefits are due to a better match between data layouts and the efficient memory access patterns supported by the hardware.

Dymaxion [6] requires users to modify applications by replacing memory transfers and array indices with the Dymaxion API. In contrast, Dymaxion++ provides a more friendly programming style with directive annotation. This relieves programmers from modifying the actual computation code. The pragma mechanism is flexible to enable, disable and configure different parameters for different devices. In fact, for the current implementation, a Dymaxion++ program will be translated at compile time into a form similar to a program developed with the Dymaxion API. In terms of the actual application execution time, there is little difference between Dyamxion and Dymaxion++ versions.

V. Memory Space Mapping

In this section, we describe Dymaxion++'s support for mapping data structures to specialized memory spaces in GPUs.

A. Mapping to Scratchpad Memory

Dymaxion++ is designed to be a high-level programming abstraction to define memory mappings. The actual implementation of the API can be accomplished in software, hardware, or both. It depends on specific hardware features and the memory hierarchy of particular platforms. In many cases, it is possible to use layout transformation in combination with memory-space mapping. For instance, the code in Listing 5 shows an example of a combined *row2col* remapping and scratchpad data mapping. We compare two possible approaches to implement this combination across different memory-hierarchy levels. Their graphical illustration is shown in Figure 4

```
unsigned int tid = get_global_id(0);
if (tid >= npts) return;
   ...
#pragma dympp device(GPU) \
        reshape row2col(feature[npts, nfts]) \
        place scratchpad(feature[npts, nfts], BLOCKSIZE)
for (int l=0; l<nfts; l++){
     ans += (feature[tid*nfts+l]-clusters[i*nfts+l])*
            (feature[tid*nfts+l]-clusters[i*nfts+l]]);
}
   ...
```

Listing 5. Combined layout transformation and memory-space mapping

1) Mapping to Scratchpad – Layout Transformation across PCI-E: In Section IV-A, we discussed that layout transformation can be implemented during PCI-E transfers by chunking data structures and launching the remapping kernel after the transfer of each chunk. After layout transformation, a desirable layout will be ready in the GPU device memory for future thread accesses. In addition, programmers can map data structures to the on-chip scratchpad memory to take further advantage of data locality and reuse for higher performance (See Figure 4(i)).

In a GPU, per-SIMD scratchpad (local data share by AMD and shared memory by NVIDIA) usually is divided into banks. These banks are organized such that successive addresses are assigned to successive banks. Therefore, any memory load or store of n addresses that spans n distinct memory banks can be serviced simultaneously. Simultaneous accesses by multiple threads to data within the same bank will cause bank conflicts.

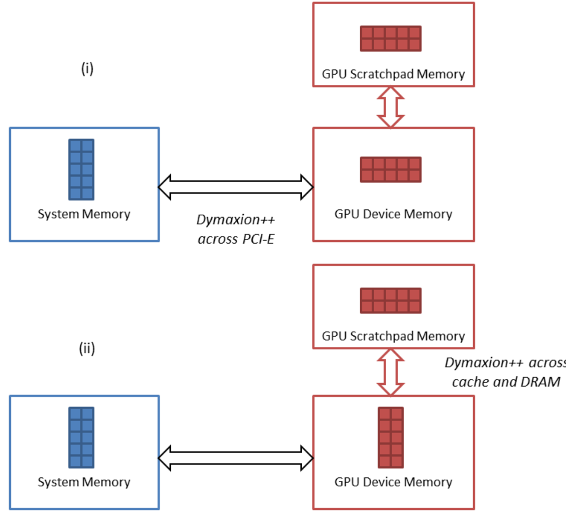

Fig. 4. Two possible implementations of combined layout transformation and memory-space mapping: 1) layout transformation across PCI-E and 2) layout transformation across the device memory and scratchpad

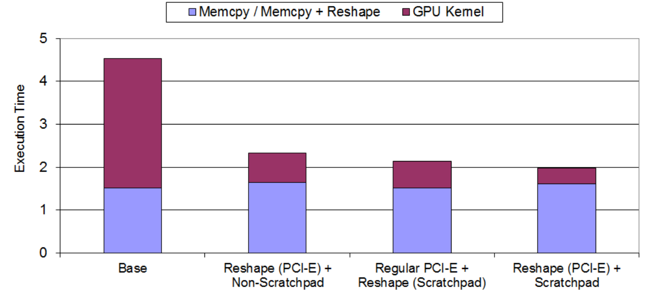

Fig. 5. Execution time breakdown of *Kmeans* due to memory remapping and PCI-E transfer (NVIDIA GTX 480). Base is the case without any layout and memory mapping optimizations. "reshape (PCI-E)" means the layout transformation is performed during PCI-E transfer. "reshape (Scratchpad)" means the layout transformation is performed during the data transfer between the device and scratchpad memory.

```
int bx =  get_group_id(0);
int tx =  get_local_id(0);
int tid = BLOCK_SIZE * bx + tx;

//load data into feature_t (row2col) in scratchpad
for(int i = 0; i < nfeatures; i++)
   feature_t[i * BLOCK_SIZE + tx]=feature[tid * nfts + i];

//determine the centroid for each data point
float min_dist=BIG_NUM;
for (int i=0; i < nclusters; i++) {
   float dist = 0, ans = 0, value;
   for (int l=0; l<nfts; l++){
       //r2c_idx transforms the index for the new array
       value =  feature_t[r2c_idx(tid, l, npts, nfts)];
       ans +=  (value-clusters[i*nfts+l]) *
               (value-clusters[i*nfts+l]);
   }
   dist = ans;
   if (dist < min_dist) {
       min_dist = dist;
       index    = i;
   }
   membership[point_id] = index;
}
```

Listing 6. An example of translated *Kmeans* GPU code after the combined layout transformation and memory-space mapping. Layout transformation is conducted during data communications between the device and scratchpad memory.

From a programmer's point of view, the requirements for efficient bank accesses are similar to those of DRAM coalescing. If the data structure is transformed between the system and device memory (by overlapping chunked PCI-E transfers and layout transformation), Dymaxion++ will produce a different data layout in the device memory. This layout is naturally desirable for scratchpad accesses as well for the most cases. In this case, the Dymaxion++ source-to-source translator will generate code to declare a shadow array in the scratchpad memory, fetch data from the device memory to the shadow array (with the same layout), and redirect memory accesses to the shadow memory. Dymaxion++ also will produce code to transform the array indices for accessing the shadow array. Both work-group and shared memory size are important for GPU occupancy and parallelism [15], [17]; we leave automatic determination of the best resource-allocation strategy as future work.

2) Mapping to Scratchpad – Layout Transformation across DRAM and Scratchpad: Alternatively, layout transformation can be implemented in another way. In a second implementation, we instead transfer a linear contiguous data region in bulk from the system memory to the GPU device memory (i.e., regular *clEnqueueWriteBuffer* and *cudaMemcpy*) without chunking. Layout transformation is performed when loading data from the device memory to the scratchpad memory (Figure 4(ii)). In this scenario, the application needs to pay the one-time cost of gathering data from non-contiguous memory blocks in the device memory into the scratchpad. If the algorithm has sufficient data reuse, this overhead can be amortized over future memory accesses and thus the application benefits from performance improvement due to more efficient layout. In addition, prior techniques [3] can be integrated with possible latency hiding, with certain threads dedicated to data prefetching and transformation.

Similarly, in this case Dymaxion++ generates code to declare a shadow array in the scratchpad memory for the target array to be transformed. When executing each work-group, data is loaded from the device memory to the shadow array in a remapped order defined by the programmer. Array indices are transformed to access the remapped array. Code Listing 6 shows the resulting code from such transformation.

Figure 5 shows the total execution times for *Kmeans* on a NVIDIA GTX480 GPU. We considered four cases for the experiments. *Base* refers to the original naive implementation without any memory remapping. In the second experiment, Dymaxion++ conducts layout transformation during the chunked PCI-E transfer, and the GPU kernel accesses the remapped data structure in the GPU device memory. The performance of *kmeans* distance kernel improves 4$\times$ due to better-coalesced memory accesses. Considering both layout remapping and PCI-E overhead, the overall execution time is approximately half of the original version. In the third experiment, data-layout transformation is performed when exchanging data between

the device memory and scratchpad memory *after* the bulk PCI-E transfer. The performance of the GPU kernel improves an additional 10% and the overall performance improves 8%. Accessing the local scratchpad memory achieves higher bandwidth and lower latency. In the last experiment (the fourth experiment), which improves the second implementation with layout transformation across the system and device memory, Dymaxion++ loads the remapped array from the GPU device memory to the scratchpad memory with the same layout. In this case, the performance of the GPU kernel improves an additional 40% and the overall performance improves 8% compared to the third case. The fourth case overlaps GPU layout transformation with PCI-E transfer, therefore the combined PCI-E transfer + layout transformation consumes a little more time than the third case which conducts regular PCI-E transfer. In addition, since the in-kernel layout transformation is conducted between the device and scratchpad memories for the third case, its kernel execution time is slower than the four case by about 70%.

We also perform the same set of experiments for *Needleman Wunsch*. The second experiment, where Dymaxion++ conducts layout during the PCI-E transfer, and the GPU kernel accesses the remapped data structure in device memory, improves performance by 23% for kernel execution and 16% for overall performance. Because the *Needleman Wunsch* kernel presents little data reuse, the performance of the third case achieves similar performance to the first case, while the fourth case's performance is similar to the second case.

Programmers can decide and specify what Dymaxion++ mechanism to use based on their knowledge of the application for potential benefits. On the other hand, GPU performance models [13] can be integrated to estimate performance benefits of different transformations.

B. Texture and Constant Memory

Conventionally, data structures must be declared explicitly to use texture and constant memories and special API functions must be applied to manage and access these buffers. Aside from the tedious process, the optimized code may not be performance portable across devices. Dymaxion++ provides the *place* construct and the *constant* and *texture* clauses for users to specify mappings to these spaces.

```
unsigned int tid = get_global_id(0);
...
if (idx <= i ){
  unsigned int index = tid * dim + (i - tid);
  #pragma dympp device(GPU) \
  reshape diagonal(score[dim, dim]) \
  place texture(ref[dim, dim])
  score[index]= max(score[index-1-dim]+ref[index],
                    score[index-1]    -penalty,
                    score[index-dim]  -penalty);

}
```

Listing 7. Map the substitution score matrix (*ref*) in *Needleman Wunsch* to texture memory

An example demonstrates a Dymaxion++ mapping to the texture memory. In the *Needleman-Wunsch* score-filling kernel, a 2-D reference array (*ref*) stores the substitution score for each value at the coordinate i and j. It is used to calculate the optimal score for alignment and is read-only during the entire program execution. We apply the *texture* clause to map the *ref* array into the texture space in addition to transformation of the *score* array. The sample code with Dyamxion++ directives is in Listing 7.

Two transformations are performed by Dymaxion++. In the case of OpenCL, Dymaxion++ translates the code to use *clCreateImage* and *clEnqueueCopyBufferToImage* for buffer management on the host side and use *read_image* to access the texture array. In CUDA, Dymaxion++ inserts code on the host side to declare a new array in the texture space and binds the global array via the CUDA API call *cudaBindTexture*. On the kernel side, it replaces the original array in the global memory space with the new texture array. *tex1Dfetch* is used to access the structure. We measure the performance on an AMD Radeon HD 7950 GPU. The texture-memory version achieves an average of 20% performance improvement compared to the original version, which accesses the device memory.

Constant memory mapping can be implemented similarly (i.e. *place constant*). In the *Kmeans* distance kernel, a small 2-D array stores all the centroids of *k* clusters. We map this array to the GPU constant memory. When conducing distance calculations, all the threads will access the same data when navigating the features of an individual centroid. For OpenCL, transformation is relatively simple; in the kernel argument, the cluster array is labeled with *__constant*. For CUDA, two transformations are performed by Dymaxion++. On the host side, it inserts the code to declare a new array in the constant space and copies the global array to the new structure via the CUDA API call *cudaMemcpyToSymbol*. On the kernel side, Dymaxion++ replaces the original *cluster* array in the global space with the new constant array. The constant memory version consistently improves the execution time by 3% on a NVIDIA GTX 480.

VI. Related Work

Prior research studied directive-based programming models for the GPU architecture. OpenACC [16] offers a mode describing a collection of compiler directives to specify loops and regions of code in standard C, C++, and Fortran to be offloaded from a host CPU to an attached accelerator. Similarly, OpenHMPP [4] allows annotations of code to offload procedures in codelets (i.e., functions to offload) onto a remote device and optimizations of data transfers. The recent OpenMP 4.0 [18] includes programmings features for offloading computation to accelerators. Szafaryn et al. [21] proposed a high-level programming framework, supporting a common codebase with translations of the code to OpenMP and PGI Accelerator API. However, these models do not include features to optimize data access patterns and layouts. hiCUDA [7] defined a directive-based API for programming NVIDIA GPUs. The C-to-CUDA [2] work describes an automatic code transformation system that generates parallel GPU code from sequential C code for regular programs. OpenMPC [12] is directive-based and provides a higher-level API built on top of OpenMP. OpenMPC implements a compiler that automatically generates

GPU code by interpreting directives and provides a variety of optimization options. The above works include data mapping of structures to specialized memories. However they do not support layout transformation or its potential combination with memory-space mapping. In contrast to these works, our work is unique in focusing on a more comprehensive API designed specifically for memory layout transformation and space mapping.

Sung et al. [20] investigated a compiler approach for layout transformation for GPU kernels, focusing on structured-grid applications. The DL [19] work studies an Array-of-Structure-of-Tiled-Array (ASTA) layout and in-place data marshaling for improving the device memory throughput for GPU. Jang et al. [10] used a mathematical model and algorithms to analyze data access patterns and target loop vectorization and GPU memory selection with different patterns. Zhang et al. [23] proposed a library to reduce irregularities in GPU programs through a level of indirection and job swapping to improve branch and memory divergence. In contrast to these works for GPUs, our work is focused on a more general, directive-based programming framework. This approach enables programmers to influence the underlying memory mapping and transformation for common access patterns without breaking the original code structure and achieving good performance portability. In addition, Meng et al. [14] proposed a hardware mechanism of dynamic warp subdivision for improving branch and memory divergence tolerance. Yang et al. [22] developed a compiler proposing a variety of techniques such as vectorization, tiling and unrolling, data prefetching, etc. Their paper mentions the use of scratchpad memory to yield better coalescing. Our framework targets a comprehensive understanding and support for data and memory remappings, and provides different possibilities for their implementations.

VII. Conclusions and Future Work

This work is a first-step preliminary study proposing a directive-based API framework, Dymaxion++, to ease GPU optimizations and improve performance portability. It complements existing GPU API designs with a focus on programming abstractions for memory layout transformation and mapping to different memory spaces. It adds pragma support for using the prior Dymaxion abstractions in a pragma-based framework, instead of requiring users to manually rewrite their programs to use the Dymaxion API. We also demonstrate the possibilities where layout transformation and memory mapping can be combined. We use several application kernels to demonstrate the use of our API. Dymaxion++ improves productivity of programmers in optimizing memory accesses. For most applications, it requires only a few lines of code changes. Our preliminary experiments show an average of 3.2× performance improvement across GPU kernels. Depending on how layout transformation is performed and degree of data reuse, the overall performance benefit ranges from 48% to 56% for *Kmeans*, and 16% for *Needleman Wunsch*. The programming effort of applying our API is trivial compared with performance gains. Future work will improve the framework with more pragma features and integration of Dymaxion++ into existing directive-based APIs such as OpenMP. We will support more layout transformation and memory mapping features for Dymaxion++, including more structures (e.g., array of structs) and optimizations across multiple nodes. We will extend this preliminary analysis with more comprehensive performance evaluation for different pragmas and applications on diverse platforms. In addition, future work will also study data layout issues for irregular algorithms, e.g., sparse matrix and graph theory problems. It would also be an interesting research to study compiler techniques which allow automated source code analysis and Dymaxion++ optimizations, thus reducing developer involvement.

VIII. Acknowledgements

This work was supported in part by NSF grant CCF-1116673. We thank the reviewers for their constructive comments.

References

[1] AMD Accerated Processing Unit (APU). Web resource. http://www.amd.com/us/products/technologies/apu/Pages/apu.aspx.

[2] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the International Conference on Compiler Construction*, Mar 2010.

[3] M. Bauer, H. Cook, and B. Khailany. CudaDMA: optimizing gpu memory bandwidth via warp specialization. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011.

[4] CAPS OpenHMPP. Web resource. http://www.caps-entreprise.com/openhmpp-directives/.

[5] 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 *Proceedings of the International Symposium on Workload Characterization*, Oct 2009.

[6] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing Conference)*, Nov 2011.

[7] T. Han and T. S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, Mar 2009.

[8] B. He, N. K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2007.

[9] Intel Many Integrated Core Architecture. Web resource. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[10] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22:105–118, 2010.

[11] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development, 49(4/5):589-604*, 2005.

[12] S. Lee and R. Eigenmann. OpenMPC: Extended openmp programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010.

[13] J. Meng, V. Morozov, K. Kumaran, V. Vishwanath, and T. Uram. GROPHECY: Projecting GPU performance from CPU code skeletons. In *Proceedings of the ACM/IEEE Supercomputing Conference*, Nov 2011.

[14] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th ACM/IEEE International Symposium on Computer Architecture*, June 2010.

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

[16] OpenACC. Web resource. http://www.ece.umd.edu/dramsim/.

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

[18] The OpenMP API specification for parallel programming. Web resource. http://openmp.org/wp/.

[19] I-J. Sung and W-M W. Hwu. DL: A data layout transformation system for heterogeneous computing. In *Proceedings of the Innovative Parallel Computing*, May 2012.

[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 *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept 2010.

[21] L. G. Szafaryn, T. Gamblin, B. R. de Supinski, and K. Skadron. Experiences with achieving portability across heterogeneous architectures. In *Proceedings of the Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, May 2011.

[22] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, June 2010.

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