DL: A Data Layout Transformation System for Heterogeneous Computing

I-Jui Sung
sung10@illinois.edu

Geng Daniel Liu
gengliu2@illinois.edu

Wen-Mei W. Hwu
w-hwu@illinois.edu

ABSTRACT

For many-core architectures like the GPUs, efficient off-chip memory access is crucial to high performance; the applications are often limited by off-chip memory bandwidth. Transforming data layout is an effective way to reshape the access patterns to improve off-chip memory access behavior, but several challenges had limited the use of automated data layout transformation systems on GPUs, namely how to efficiently handle arrays of aggregates, and transparently marshal data between layouts required by different performance sensitive kernels and legacy host code. While GPUs have higher memory bandwidth and are natural candidates for marshaling data between layouts, the relatively constrained GPU memory capacity, compared to that of the CPU, implies that not only the temporal cost of marshaling but also the spatial overhead must be considered for any practical layout transformation systems.

This paper presents DL, a practical GPU data layout transformation system that addresses these problems: first, a novel approach to laying out array of aggregate types across GPU and CPU architectures is proposed to further improve memory parallelism and kernel performance beyond what is achieved by human programmers using discrete arrays today. Our proposed new layout can be derived in situ from the traditional Array of Structure, Structure of Arrays, and adjacent Discrete Arrays layouts used by programmers. Second, DL has a run-time library implemented in OpenCL that transparently and efficiently converts, or marshals, data to accommodate application components that have different data layout requirements. We present insights that lead to the design of this highly efficient run-time marshaling library. In particular, the in situ transformation implemented in the library is comparable or faster than optimized traditional out-of-place transformations while avoiding doubling the GPU DRAM usage. Third, we show experimental results that the new layout approach leads to substantial performance improvement at the applications level even when all marshaling cost is taken into account.

Keywords

GPU, Data Layout, Marshaling

1. INTRODUCTION

The OpenCL standard [3] promises portability of high performance heterogeneous parallel computing applications across a wide variety of CPU and GPU hardware. While vendors such as AMD, Intel, IBM, and NVIDIA have largely achieved functional portability of OpenCL applications to date, there has been little reuse of OpenCL application kernels across hardware platforms in practice. A major problem that hinders the reuse of kernels is their performance sensitivity to the diverse memory layout requirements of the underlying hardware. The root of the problem is the constantly increasing disparity between DRAM and processor speeds [14], which compels modern memory system designers to employ wider DRAM bursts and a high degree of memory interleaving to create sufficient bandwidth to supply operands to the numerous processing elements.

Latency-optimized CPUs with large amount of on-chip cache memories use long cache lines and deep memory channel queues to reshape transactions to the memory system and achieve high utilization of the memory bandwidth. As long as the data sets fit into the cache, the achievable bandwidth of data accesses is largely insensitive to the access patterns. As a result, CPU data sets tend to assume layouts that follow the natural organization used in external data file. For example, if each element of an aggregate data set consists of several values, such as the RGB values of a color pixel, the values for each data element are laid out in consecutive memory locations, which is consistent with most natural file formats of video cameras. Such layout is commonly referred to as the Array-of-Structure (AoS) layout.

Throughput-oriented many-core GPU systems tend to have much less on-chip cache memory, if any, per parallel execution thread when compared to their CPU counterparts. For example, the NVIDIA GTX480 GPU has a relatively small cache capacity per thread (only 34 bytes of L2 cache memory per thread, given 1536 threads per SM, 15 SMs, and 768KB shared L2 cache). The purpose of the last-level cache is to consolidate accesses from parallel threads into fewer DRAM requests rather than to support temporal reuse by capturing the working sets. Therefore, the achievable data access bandwidth is much more sensitive to the access patterns of the massive number of simultaneously executing threads. As a result, NVIDIA GPUs show strong benefit from data layout adjustments that minimize the number of cache lines used by simultaneously executing threads. In the pixel example, NVIDIA GPUs tend to prefer a data layout where all the R values of the pixels processed by simultaneously executing threads are in consecutive locations, followed by G values and then followed by B values. Such layout is commonly referred to as the Structure-of-Arrays (SoA) layout.

In statically typed languages like OpenCL and its base language C, the size of each (aggregate) field of a structure must be known at compile time. This makes it extremely

difficult, if not impossible, to declare SoA types and pointers for dynamically allocated buffers where the size of each field (array in this case) in the structure is unknown until run time. Unfortunately, the dynamically allocated buffers are the main use mode of bulk data in OpenCL kernels. As a result, programmers tend to break up the structure and simply use discrete arrays after they transform the layout by hand. We will refer to this approach as as the discrete arrays (DA) layout. However, for other GPU architectures such as the ATI Evergreen architecture [2], naïve conversion to DA layout can even hurt the performance. In the ATI Evergreen architecture, this is due to its VLIW-based design which favors short vectors of DRAM accesses by each work item, and relatively simpler design of memory interleaving comparing to NVIDIA architectures.

There are three major challenges presented to programmers in managing data layout for heterogeneous parallel computing. First, due to the diverse layout preferences of CPUs and different types of GPUs, neither AoS nor DA can satisfy the needs of all OpenCL hosts and devices. Any data layout chosen by the programmer will likely perform poorly for some parts of the application on some types of devices. This paper presents an approach that intelligently maps and re-maps the data structure used in application kernels into the most suitable layouts for underlying GPU architectures in order to achieve good off-chip memory access efficiency. Similar to the scheme proposed by Sung et al. [16] the system determines the layout based on the indexing pattern used by kernels to access aggregate data structures. The contribution of this work is a new layout arrangement that can be auto-tuned to match the needs of diverse architectures, practically eliminating the need for multiple kernel versions as far as data layout is concered.

Second, with programmer managed data layout, the programmer also must reconcile the discrepancy between the host code data layout and the transformed kernel data layout. In practice, it is often infeasible to convert the entire host code due to its size and the I/O library components that may be only available in binary form. If the programmer chose to have different layouts for a data structure in different parts of the application, he/she needs to manually create marshaling routines to convert between these layouts at run time when the application transition from one part to another. As we will show in the paper, efficient, high-throughput data marshaling is a challenging endeavor and a perfect candidate for highly optimized libraries. The system should provide default, transparent and efficient support for data marshaling.

Third, GPU DRAM capacity is usually only a fraction of their CPU counterparts. Naïve, out-of-place data conversion can easily double the memory footprint. In some cases such as large numeric applications, this can be a prohibitive factor. It is highly desirable to perform marshaling in situ without requiring additional memory.

2. CONTRIBUTIONS

In order to address these three challenges that have hindered the practical use of previous layout transformation approaches, we developed three innovative mechanisms into DL. First, we propose ASTA (Array-of-Structure-of-Tiled-Arrays) to enable both high-performance accesses in diverse architectures as well as low cost marshaling from/to array-of-structures and struct-of-arrays. We then generalize ASTA to a tiled transposition layout that is applicable on tall arrays and some sparse matrix formats. Second, novel GPU in-place marshaling algorithms are also developed as part of the DL framework. Third, we implemented an automatic data marshaling framework, as a run-time library that transparently and efficiently marshal data to accommodate application host code and kernel components that require different data layout arrangements.

3. BACKGROUND AND MOTIVATION

In following section we will discuss issues blocking the exploitation of memory parallelism on the GPUs.

3.1 Array-of-structure v.s. Structure-of-Array

Having coalesced memory access has long been advocated as one of the most important off-chip memory access optimizations for modern GPUs. However, numerical solvers for many physical problems such as CFD (computational fluid dynamics) involves solving multiple related physical properties in discretized space. Naturally, these properties can be mapped into structures and then grouped into an array, in which each GPU thread accesses its corresponding structure instance. The OpenCL kernel `AoS` in Listing 1(line 6–9) is a simplified case showing this usage. Note in OpenCL each work-item (thread) is assigned uniquely an index, which can be obtained through the `get_global_id` intrinsic call.

```
1  struct foo{
2    float bar;
3    int baz;
4  };
5
6  __kernel void AoS( __global foo* f) {
7    f[get_global_id(0)].bar*=2.0;
8  }
9
10 __kernel void DA(__global float *bar,
11   __global int *baz) {
12   bar[get_global_id(0)]*=2.0;
13 }
14
15 struct foo_2 {
16   float bar[4];
17   int baz[4];
18 };
19
20 __kernel void ASTA(__global foo_2* f) {
21   int gid0 = get_global_id(0);
22   f[gid0/4].bar[gid0%4] *=2.0;
23 }
```

Listing 1: AoS, Discrete Arrays, and ASTA

It is commonly assumed that the AoS layout of such data structure degrades the performance by creating non-unit-stride access across GPU work-items (or threads in CUDA terms) in the same wavefront (or warp in CUDA terms). A commonly applied transformation is to manually convert it to discrete arrays (DA). In this example, one declares a float array to hold all "`float bar`"s across structure instances in the array; another int array for all "`int baz`"s. This is to work around a limitation of mainstream GPGPU programming models that are derived from C: structure

types do not support variable-sized member arrays in general. So programmers usually have to implement aggregates of dynamically-allocated array into discrete arrays, one for each former structure member. This is shown in the kernel DA in Listing 1 (line 10–13).

Another practical option, also mentioned by Che et al. [5], is applicable when all members are of the same (scalar) type: replacing the structure by an additional dimension and use hard-coded indices (possibly using preprocessor macros or enumerations) for each "member". This effectively degenerates SoA to a multidimensional array of the same scalar type. Through a transposition, one can move the named indices to the highest dimension. Note that while DA and this approach are different ways of getting around the limitations of a statically typed language, Che's approach and DA are similar in their final layout. For the rest of this paper, we will use DA to broadly refer to both Che's approach and DA.

Figure 1 shows the average time for accessing a `float` data element of a micro-benchmark. In the microbenchmark, each work-item works on one of a million of structure instances in an AoS array. Work-item with global ID i accesses the i-th structure instance. Each work-item computes sum reduction over all members in that structure instance. The sum is then duplicated into all members of the corresponding instances of another array-of-structure. The duplication gives the benchmark balanced number of loads and stores. Giving the loads and stores the same level of influence on the measured cost. This benchmark does very little computation so it is obviously memory bound. For each architecture, a transformed version (from AoS to DA) is presented to show the relative memory bandwidth gain.

The results from the NVIDIA architecture match the conventional wisdom of GPU data layouts: the cost of accessing the AoS grows almost linearly as the structure size increases. A reasonable explanation is that as the size of the structure increases, the stride of the accesses within each wavefront also increases. This increases the portion of each DRAM burst that is discarded by the memory access unit. The Discrete Array curve shows that the DA layout preserves the efficiency of DRAM accesses as the size of the structure grows. Surprisingly, on the ATI architecture the AoS layout performs better than the DA layout for structures smaller than 14 floats. There seems to be a buffer and/or a VLIW instruction schedule that allow more parts of each DRAM burst to be utilized. This means that for ATI architectures, moderately sized AoS is the better choice over DA. We believe that after 16 elements, the working set sizes of AoS buffer of this particular benchmark exceed the cache sizes on that particular architecture.

Figure 1 shows that choosing a single layout for portable performance is not trivial. Naïve conversion of all GPU kernels to discrete arrays might work well for NVIDIA GPUs, but it is not the best choice for ATI GPUs. Without a good programmer-level strategy for all architectures, the programmers will always be compelled to write multiple versions of kernels in order to get good performance on each architecture. We show such a strategy in this paper.

3.2 In-place Layout Conversion

Consider the layout of array `f` which is passed to kernel AoS in Listing 1, Line 6. Assume that the programmer has changed to kernel DA in Listing 1, line 11. Since array f is still in AoS form on the host side, it needs to be marshaled into the new DA form for use by the new kernel. To convert array f to a DA layout in GPU, one approach is to launch a kernel with $2n$ work-items. Each work item uses its index to load a distinct f element, one of the two scalar members `bar` and `baz`, into its register. This is illustrated in Figure 2. All work items then perform a barrier synchronization to ensure that everyone has finished loading its assigned element. After the barrier, all work items store the loaded value to new locations in the new discrete arrays, as shown in Figure 2.

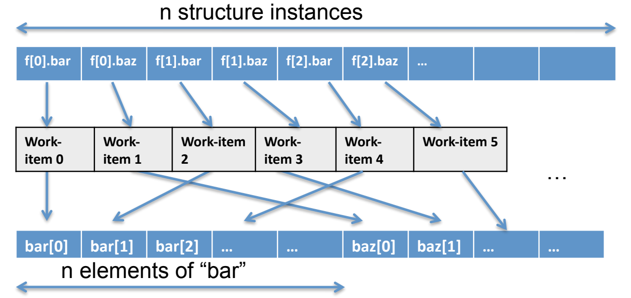


Figure 2: Converting Layout of Array F

There are however two problems. First, the array size (n) is usually large for GPU workloads, but the scope of barrier synchronization in current GPU architecture is fairly small; in general GPU architectures do not support global barriers across work-groups, each of which usually consists of at most 1024 work-items (fine-grained threads) out of tens of thousands of total work-items. This means a straightforward GPU-based in-place marshaling kernel would not scale much beyond 1024 work-items. If we see the problem of converting array f to SoA as transposing an 2-by-n column-major matrix in-place, then in this approach the scope of barrier synchronization must be large enough to cover any cycles in the transposition process.

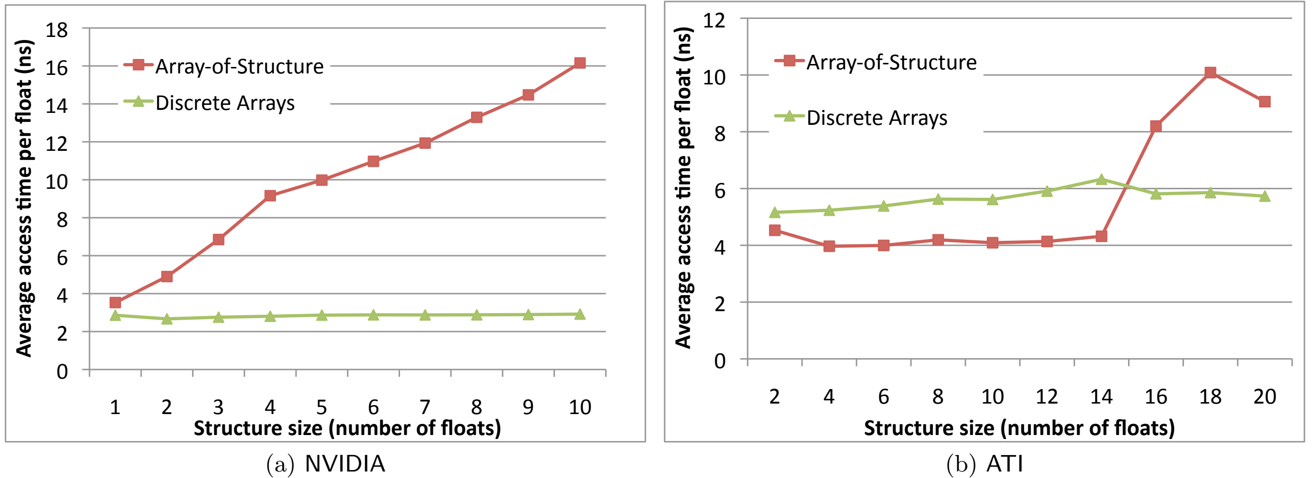
Mathematically, in-place transposition is a permutation that can be factored into a product of disjoint cycles [11]. Assume that A is a m-rows-by-n-columns array ($m \times n$ for brevity), where $A(i, j)$ is the element in row i and column j. (In the following text, when we refer to a element in a row-major array, we use C-like syntax like `A[i][j]`; when we refer to an element in a column-major array, we use FORTRAN-like syntax like `A(i, j)`.) In a linearized column-major layout, $A(i, j)$ is in offset location $k = i + jm$. The transposed array A' is an n-rows-by-m-columns array, and $A(i, j)$ at offset k is moved to $A'(j, i)$ at $k' = j + in$ after transposition. The formula for mapping from k to k' is:

- $k' = kn \mod M$ if $0 \leq k < M$
- $k' = M$ if $k = M$

where $M = mn - 1$. For transposing $m \times n$ row-major array, the formula is:

- $k' = km \mod M$ if $0 \leq k < M$
- $k' = M$ if $k = M$

We can generate a "chain" of mapping by starting with an arbitrary offset k_1, mapping it to k'_1 and using k'_1 as k_2

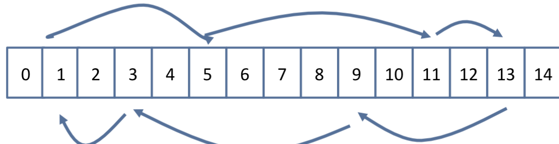


(a) NVIDIA (b) ATI

Figure 1: Speedup of Discrete-Array over AoS layout on a simple reduction kernel

for the next mapping. For example, we can use a column-majored 3×5 matrix transposition example shown in Figure 3. We start with $k_1 = 1$ (the location of $\mathtt{A}(1,0)$) and map it to $k'_1 = 5$ (the location of $\mathtt{A}'(0,1)$). We can then use $k_2 = 5$ (the location of $\mathtt{A}(2,1)$) and map it to $k'_2 = 11$ (the location of $A'(1,2)$); the chain element at location 5 will be shifted to location 11, and the element at location 11 will be shifted to location 13, and so on. Eventually, we will return to the original offset 1. This gives a cycle of `(1 5 11 13 9 3 1)`. For brevity, we will omit the second occurrence of 1 and show the cycle as `(1 5 11 13 9 3)`. The reader should verify that there are five such cycles in transposing a 5×3 column-majored matrix: `(0) (1 5 11 13 9 3)(7)(2 10 8 12 4 6)(14)`.

A (0,0)(1,0)(2,0)(0,1)(1,1)(2,1) (0,2)(1,2)(2,2)(0,3)(1,3)(2,3)(0,4)(1,4)(2,4)



A' (0,0)(1,0)(2,0)(3,0)(4,0)(0,1) (1,1)(2,1)(3,1)(4,1)(0,2)(1,2)(2,2)(3,2)(4,2)

Figure 3: Converting Layout of Array F

An important observation is that an in-place transpose algorithm can perform the data movement for these five sets of offset locations independently. This means that we only need to synchronize the data movement within each cycle.

Unfortunately, the number of cycles and the length of each cycle vary with problem size and there is in general no locality between elements in a cycle [13] in in-place transposition. Note for square matrices, the size of a cycle is either 1 (diagonal) or 2 (other elements), but in the case of Array-of-Structure, the aspect ratio is usually not 1:1, as the number of elements in a structure is usually much smaller than the total number of structure instances. We will address this point further in Section 4.4.

4. APPROACH

The proposed approach consists of three parts: the ASTA layout, in-place marshaling from AoS and DA to ASTA, and the design of a dynamic runtime marshaling library for OpenCL.

4.1 The ASTA Layout

Given an AoS layout, we can convert T adjacent structure instances into a mini SoA. We call this scheme Array-of-Structure-of-Tiled-Array (ASTA). In Listing 1, the structure type in Lines 15–18 and kernel ASTA shown in line 20 is an example of ASTA. Note the `struct foo_2` is derived from struct foo by merging 4 instances of `struct foo` and generate a "mini SoA" out of each merged section. Effectively, each scalar member in struct foo is expanded to a short vector in `struct foo_2`. We call the length of this short vector (T) the coarsening factor of the ASTA type. The short vector is called a *tile*. Usually the coarsening factor is at least the number of work-items participating in memory coalescing. ASTA improves memory coalescing while keeping the field members of the same original instance more closely stored, and is thus potentially useful to reduce memory channel partition camping due to large strides [16, 15].

The AoS layout can be considered as an $M \times S$ array where S is a small integer in row-major layout. In this way, DA is $S \times M$. Similarly, ASTA is similar to $M' \times S \times T$ where $M = M'T$.

At a high level, marshaling from AoS to ASTA is similar to transpose M' instances of small $T \times S$ matrices. Whereas marshaling from DA to ASTA is similar to transpose a matrix of $S \times M'$ of T-sized tiles.

We propose three algorithms here to facilitate efficient in-place marshaling. For AoS to ASTA, when $T \times S$ is small enough, a barrier-synchronization-based approach is proposed. When $T \times S$ is larger (but still not as large as a full matrix transposition), a fast cycle-following approach that exploits locality within an ASTA instance is proposed. For DA to ASTA, we exploit the fact that the T can cover one or more cache lines, so there is good locality when moving tiles.

4.2 In-place Conversion from AoS

One of the benefits of ASTA that cannot be easily accomplished otherwise by discrete arrays is allowing in-place marshaling. Let us consider an Array-of-structures, with all structure members being of the same size. With this, Array-of-Structure to Discrete Arrays transposition is like transposing this tall array and treating the starting address of each row after transposition as different arrays.

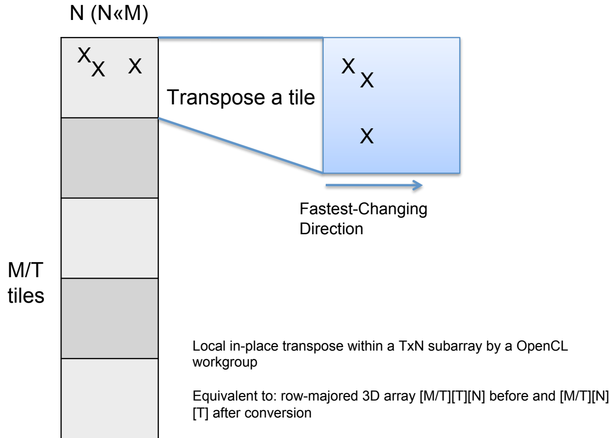


Figure 4: In-Place Marshaling of a Tall Matrix

For ASTA, instead of performing a full "transposition" (difficult for non-square matrices if performed in-place) on the entire array (thus converts to SoA), we use a tiled transposition that only transposes within subarray of the original array (i.e. an ASTA instance). This allows fast in-place marshaling as transforming each tile can be performed entirely by an OpenCL work-group, and enables high-performance coalesced access to the transformed layout. An example of converting a tall matrix (which is also the case of a sparse matrix represented in ELL [1, 4] format) is shown in Figure 4. This allows coalesced accesses along the column direction in the example being vectorized in T-sized tiles. The size of T is usually between 16 and 64 across GPU architectures for memory coalescing. Note T is equivalent to the coarsening factor in ASTA.

If each work-item is assigned to a structure element in AoS, the forward marshaling kernel from AoS to ASTA can be implemented in three steps (which is the same as the scheme in Figure 2, except that n is always equal to the coarsening factor):

1. Each AoS element is loaded into a private register in the work-item
2. A barrier synchronization with memory fence (`barrier(CLK_GLOBAL_MEM_FENCE)in OpenCL`) to make sure all loads by all work-items in the same group is finished at this point
3. The data held in the registers are stored to the target ASTA fields in-place

This approach assumes that the coarsening factor times the structure size is no larger than the maximum number of work-items per work-group supported by the system. For small AoS or tall arrays, this generally holds. One straightforward work-around to the work-group size limitation is to use the local memory as the temporary storage instead of registers. But in general, the tile size cannot exceed the size of on-chip memory accessible to a work-group, be it the local memory or the register file. A more general approach to addressing other cases is addressed in section 4.4.

4.3 In-place Conversion from DA

Consider an array in DA layout, say G[2][n] in row-major order and we define `G[0][0..n-1]` for storing element "foo" and `G[1][0...n-1]` for storing element "bar", both of the same type. Then if the array is accessed with expression `G[0][get_global_id(0)]` in OpenCL kernels, work-items in the same wavefront would have coalesced memory accesses. However, as we will show later, the large stride (n-elements) between `G[0][i]` and `G[1][i]` can lead to suboptimal memory performance on some GPU architectures, especially the current ATI GPUs. As observed by Sung et al [16], this is related to partition camping problem especially when n is power-of-two. Converting to ASTA can reduce the strides and hence partition camping in this case.

However, as we mentioned earlier, this is essentially transposing at a scale large enough that cannot be done easily using the same approach for converting AoS to ASTA. Fortunately, for the n values that are multiples of ASTA tile size T, there are efficient solutions.

First, if n is multiple of tile size T, say $n = n'T$, then converting the example array G to ASTA is essentially taking a row majored array `G[2][n'][T]` and transpose on the top two dimensions, leading to `G'[n'][2][T]`. Another way to view this conversion is to consider the problem as transposing a 2D row-majored array `G[2][n']` of elements, each consisting of T consecutive elements in the original array. In the following discussion, we will refer to each such "element" as a tile. As we discussed before, we can identify the cycles in the transposition and perform the data movement in each cycle independently.

Consider an example, say $n' = 5$. Then the cycles in 2x5 row-major transposition is `(0)(1 2 4 8 7 5)(3 6)(9)`. We can perform the data movement for the 4 cycles independently. Note that since we are shifting T-sized tiles, not isolated elements in this case, we have reasonably good locality for T larger than the wavefront size, by having a number of work-items shifting data values in each tile in a coalesced manner.

A simple solution is to have each cycle assigned to a work-group and having the work-group shift the tiles in its assigned cycle sequentially. This is a straightforward GPU parallelization of the cycle-following algorithm IPT [10] but works at the tile level; we call this P-IPT. However, since both the number of cycles and their lengths varies widely across different problem sizes, there is a nontrivial load imbalance problem. For example, the largest cycle in our 2X5 example has 6 tiles whereas the smallest cycle has only one tile. Our GPU implementation of this simple approach sees drastic performance variance from 0.44GB/s to 13.65GB/s on NVIDIA Fermi, on the same array with different tile sizes (16 and 64 respectively), which changes the aspect ratio of array in terms of tiles and thus the cycles for moving tiles.

To solve this load imbalance problem, we would like to further parallelize the data movement in a single cycle by having multiple work-groups to collaboratively move the tiles

within a long cycle. However, we need to coordinate the activities across multiple work-groups so that no tile would be overwritten prematurely.

To coordinate the shifting between tiles working on the same tile, we employ atomic operations and a MN'-bit auxiliary storage to mark the finished tiles. The outline of this approach (Parallel-Tile-Transpose-Within-and-Across-Cycles (PTTWAC)) for each work-group is shown in the Algorithm 1.

Algorithm 1: Parallel-Tile-Transpose-Within-and-Across-Cycles (PTTWAC)

```
Input: A: an M × N' array of T-sized tiles
Output: A: an N' × M array of T-sized tiles
Data: done : M × N'-bit array initialized 0 private to
      each work-group. A bit i is set if the values of
      tile i have been computed (not necessarily
      stored).
Data: R1,R2: private registers to each work-item;
      local_id: ID of each work-item within the
      work-group
Launch: MN' − 1 work-groups that execute
asynchronously
foreach workgroup i of size T in MN' − 1 workgroups
do
    if done[i] ≠ 0 then
        return
    next_in_cycle ⟵ (i * M)%(M * N' − 1)
    if next_in_cycle == i then
        return; //no need to shift
    /* Cooperatively load a tile i of A          */
    R1 ⟵ A[i][local_id]
    while true do
        /* Cooperatively load a tile at
           next_in_cycle                          */
        R2 ⟵ A[next_in_cycle][local_id]
        if local_id = 0 then
            if atomic_set(done[next_in_cycles]) ≠ 0
            then
                Terminates all work-item of the
                workgroup
        A[next_in_cycle][local_id] ⟵ R1
        R1 ⟵ R2
        next_in_cycle ⟵ (next_in_cycle * M)
        mod (M * N' − 1)
```

Note the `atomic_set()` operation attempts to set the bit specified by the first argument in global memory and return the original value of that bit.

Let us take the earlier example on transposing an 2x5 row-major array, and 9 work-groups are launched. Assuming only 4 work-groups can be scheduled due to hardware resource limitations in the following scenario; also recall the cycles are `(0)(1 2 4 8 7 5)(3 6)(9)`:

1. Work-group 0, 1, 2, 3 are scheduled. Then work-group 0 terminates without copying. Work-group 1, 2, 3 load tiles 1, 2, 3 into their R1, load tiles 2, 4, 6 into their R2, atomically set done[2], done[4], done[6], and then store their R1 to tiles 2, 4, 6.

2. Work-group 4 is scheduled as work-group 0 quits, and found tile 4 is shifted already (done[4] is set). Work-group 4 also quits. Work-group 1, 2, 3 load tile 4, 8, 3, and work-group 1 finds its next tile (4) is already shifted, so it quits. Work-group 2 and 3 atomically set done[8] and done[3] and store to their next-tile-in-cycles 8 and 3.

3. Work-group 5 and 6 are scheduled for execution since 2 work-groups quit in previous step, and work-group 6 terminates immediately as tile 6 is shifted at step 2. Work-group 7 is then scheduled. Work-group 7 and 5 shift tile 7 and 5 to tile 5 and 1.

4. All tiles are now shifted; the remaining work-group 2, 3, 5, 7 quit.

In this scheme, the parallelism in shifting elements of the same cycle is exploited: at step 3 above, work-groups 2, 5, 7 are working on the largest cycle in parallel, greatly improving the speed. The spatial overhead is small as we only need one bit for each tile: even for architectures that only supports integer atomic operations, the *done* array only takes MN'-words overhead of compared to the original array. For T=64, that means roughly 1.5% extra space. On architectures allowing atomic bit operations the cost is less than 0.05%.

Qualitatively speaking, because of the randomness of positions of tiles in the same cycle, sequentially-scheduled work-groups may work on far-apart portions in the same cycle (like how work-groups 2, 5 and 7 in step 3 above worked on tiles 8, 5 and 7. Intuitively, the longer the cycles, the larger the number of work-groups will likely be working on them; thus balancing the loads of work-groups dynamically.

4.4 Extending PTTWAC to support large AoS to ASTA transformation

As briefly mentioned in Section 4.2, when the size of an ASTA instance is too large in AoS to ASTA conversion, exceeding the maximal number of work-items allowed (or number of registers in general) within the scope of one barrier synchronization, some form of cycle-following algorithm should be used to avoid barrier synchronization across work-groups. The PTTWAC algorithm can be slightly modified to handle this case. That is, have one work-group to work on an $M \times T$-sized ASTA instance, and launch a fix number of threads, say B, and each work-item to work on shifting $M \times T/B$ scalar value inside an ASTA instance. M is the number of elements of the original structure type.

Qualitatively, this scheme effectively constrains any parallelized cycle during transposition within an ASTA instance, so the working set for each work-group is also within an ASTA instance. This avoids poor locality observed in general in-place transformation. Also, since the frequency of atomic operations will be much higher, we use OpenCL local-memory (on-chip scratch pad) to store the bit array done.

4.5 Integrate the layout transformation and marshaling

In the DL system, the need of specializing the marshaling kernels based on structure type and coarsening factor is accommodated on-the-fly as an integrated part of the kernel transformation process, and then invoked by the marshaling runtime. This is described in following sections.

While the data marshaling kernels described in this paper could be and will be exposed the the OpenCL developers as a library of efficient layout adjustment functions, they can provide even more value as part of a transparent data layout transformation system. In the DL system, the need of specializing the marshaling kernels based on structure type and coarsening factor is accommodated on-the-fly as an integrated part of the kernel transformation process, and then invoked by the DL runtime. As a result, the data marshaling activities can be totally transparent to the host code. This is described in following sections.

5. KERNEL TRANSFORMATION AND RUNTIME MARSHALING

To automatically reconcile layout differences between transformed kernel at runtime, the system must be able to:

- Recognize the access pattern of the kernel
- Transform accesses to buffers used by the kernel if necessary
- Inform the runtime that the buffers need to be marshaled into desirable layout before invoking the kernel

At runtime, the runtime marshaling library must be able to:

- Marshal the kernel right before the kernel launch
- Invoke the inverse marshaling kernel right before the transformed buffer is copied back to host

The system assumes that the dimensionality of the buffer is rectangular. With this, it is possible to decouple the transformation and marshaling. Here is a step-by-step description of the process using the AoS kernel in Listing 1. Let us for now assume the kernel is transformed statically.

5.1 Step 1. Kernel transformation

In this step the kernel is analyzed and transformed. We assume the user exposes the dimensionality of buffers to the tool in the annotation in kernel source as shown in listing below. The static transformation tool parses the code and decides to transform it to ASTA, insert a new coarsened type and change the kernel code accordingly. The layout heuristic is simple:

- Convert AoS to ASTA if detected on both architectures
- Convert DA to ASTA for ATI architecture if the structure is larger than a threshold of 10 floats (found by microbenchmarking)

To ease reading, the threshold is set to 1 float in the following example. The transformed code is shown in the second half of the Listing 5.1. The annotations are on line 5 and 18; the code modified is one line 7, 18 20 and 21.

```
1  struct foo{
2    float bar;
3    int baz;
4  };
5  //DL: AoS: f[global_size(0)]
6  __kernel void AoS( __global foo* f) {
7    f[get_global_id(0)].bar*=2.0;
8  }
9  struct foo{
10   float bar;
11   int baz;
12 };
13 struct foo_2{
14   float bar[4];
15   int baz[4];
16 };
17
18 //DL: AoS: f[global_size(0)] AOS2ASTA_foo
19 __kernel void AoS( __global foo* f) {
20   offset_t t1 = get_global_id(0);
21   f[t1/4].bar[t1%4]*=2.0;
22 }
```

After transformation, the tool inserts necessary information for the runtime. In this case, the runtime needs the exact values that is available at the moment the kernel is launched; i.e. the values of the dimensionality of the transformed buffer, and the marshaling kernel to invoke. For the example, the tool generated a marshaling kernel called `AOS2ASTA_foo` into a separate file that is accessible to the DL runtime and append its name to the annotation so that at runtime, the marshaling kernel can be located.

5.2 Step 2. Run-time marshaling for OpenCL

An important feature of DL is to allow the host code to remain unchanged when using a kernel with a transformed data layout. It also supports an interface for incrementally transforming the host code components to use transformed data layouts. This allows a development team to modify only the performance-critical parts of an application to use the new data layout and avoid the pitfall of requiring massive, wholesale changes to the entire application. In fact, we envision that most of the host code will continue to use the original data layout for many applications. DL achieves this by supporting a dynamic marshaling mechanism that takes advantage of the OpenCL memory model.

OpenCL requires explicit data transferring/remapping routines to transfer data between host and device sides when invoking a kernel. Plus, OpenCL memory buffers at the device side are explicitly created and managed through a run-time library interface. The DL memory marshaling system has to keep the semantics of the OpenCL memory model and transparently insert marshaling calls only when necessary.

The observation here is that we can infer the dimensionality and layout of the OpenCL memory buffer if it is passed to a kernel that has special marshaling requirement annotated in the source by static transformation.

We use library interposition to hijack OpenCL library calls from the user. For each transformed kernel K, each argument i is augmented with $K_i \in T \times E^{\mathbb{N}} \times M$ derived from user annotation, where:

- T = Element Type $\cup$ {NIL}
- E: a symbolic expression that defines the size of each dimension.
- $M = \Gamma \cup \{\xi\}$. ξ means the layout is not transformed; Γ is the set of all layout transformations in this application. We represent a layout transformation as a pair

of handles to kernels generated by the runtime, one for converting from the original to the transformed layout and one for converting back. An example of such pair could be: (`AOS2ASTA_foo`, `AOS2ASTA_foo_inverse`). This specifies the requirement of that argument as well as the marshaling kernels to invoke.

At runtime, each OpenCL memory buffer is augmented with a tuple $S \times R^{\mathbb{N}} \times K$, where:

- $S = \{\text{Uninitialized}\} \cup$ M: S specifies the current data layout of the buffer.
- $R^{\mathbb{N}}$: the actual dimensionality of this buffer, where n is the number of dimensions of this buffer from K
- K: Last kernel argument this buffer has bound to.

At kernel launch time, the DL runtime evaluates each K_i to deduce actual dimensionality and set the corresponding R. For the example this would be $[global_size(0)]$; the corresponding R_i is passed to the marshaling kernel so that the buffer is correctly marshaled.

So, let us take the example above, and assuming the kernel is launched on 1024 work items. When the kernel K's annotation is parsed by DL runtime, the argument descriptor K_0 of its only argument is:
$\langle T : \texttt{foo}, n : 1, E : \texttt{global_size(0)}, M : \texttt{AOS2ASTA_foo}\rangle$

When a freshly initialized OpenCL buffer is passed as `f` to the kernel, it is augmented dynamically by DL runtime as:
$\langle S$: ξ, R: the allocated buffer size, K: $K_0\rangle$

When the kernel actually launches, R is evaluated to be 1024 based on `E=global_size(0)`. Then the DL runtime identifies a mismatch between `S=`ξ and `T=AOS2ASTA_foo` according to K_0. So then the marshaling kernel corresponding to the transformed layout (`AOS2ASTA_foo`) is dynamically compiled and launched with 1024 work-items. After marshaling kernel completes, the buffer is augmented as:
$\langle S : \texttt{AOS2ASTA_foo}$, R: 1024, K: $K_0\rangle$

The kernel K is then launched with the buffer in expected layout.

Should the buffer is later copied back to host code, then the inverse marshaling kernel for layout `AOS2ASTA_foo` is launched based on the descriptor status right before the actual copying occurs, and the S would be reset to ξ . If the buffer is used again by either the same kernel or another kernel with the same T and evaluates to the same R, the marshaling is avoided. If there is a mismatch between S and T and $S \neq \xi$, then we conservatively marshal the buffer back to $S = \xi$ then to T.

6. RESULTS

The following OpenCL benchmarks are used:

- LBM: a computational fluid dynamics solver using lattice-Boltzmann method
- SpMV: a sparse matrix-vector-multiplication kernel in ELL layout; each row is stored consecutively.
- Black-Scholes: an option-pricing algorithm

LBM and Black-Scholes are dense AoS layout codes whereas SpMV represents tall arrays constructed from sparse datasets. The first two benchmarks are from the Parboil Benchmark Suite; the last benchmark is adapted from NVIDIA OpenCL SDK.

For the SpMV benchmark, since the performance of layout conversion for DA to ASTA could depend on the exact dimensionality of the dataset, we use the following datasets listed Table 1.

Table 1: Test problem for SpMV benchmark and DA-ASTA in-place marshaling

Problem	Description	Size	Max. # nonzero columns
bcsstk18	R.E. Ginna Nuclear Power Station	11948× 11948	40
e40r000	Driven cavity,40x40 elements, Re=0	17281× 17281	62
bcsstk31	Stiffness matrix for automobile component	35588× 35588	197
bcsstk32	Stiffness matrix for automobile chassis	44609× 44609	215
s3dkq4m2	Finite element analysias of cylindrical shells	90449× 90449	59
conf6.0-00l8x8-8000	Quantum Chromodynamics	49152× 49152	39

Note that in ELL, the storage requirement for a matrix is the number of rows times the maximum number of nonzero columns.

6.1 Application Results

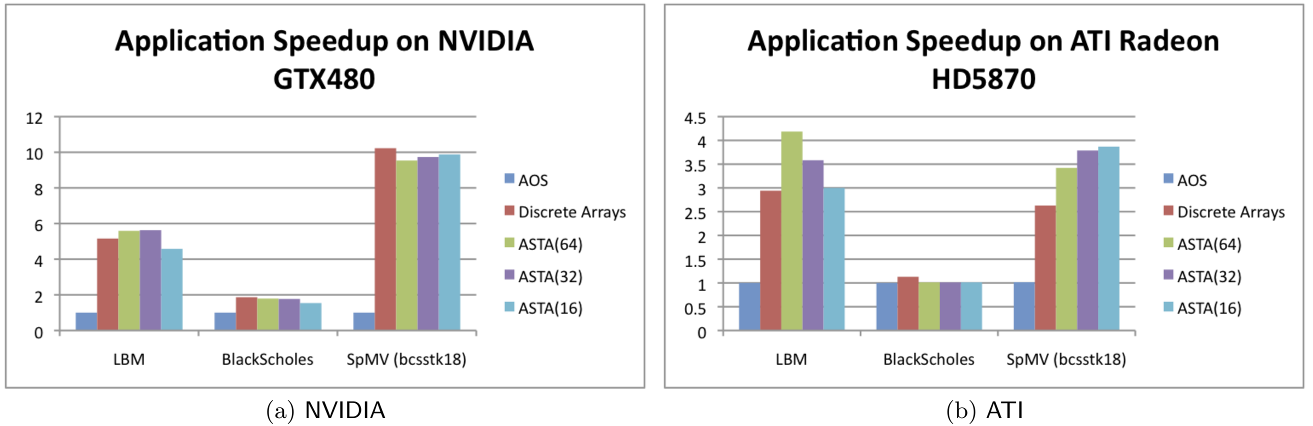
6.1.1 *Performance of Layouts*

Figure 5 shows the performance of ASTA layout as well as the generalization of tiled transposition on sparse matrix-vector multiplication (SpMV) on NVIDIA GTX480 GPU. For LBM benchmark, both the Discrete Array transformation and ASTA are able to boost the performance by more than 4X (on NVIDIA) and roughly 3X (on ATI) if the marshaling cost is fully amortized. However, the ASTA layouts on both ATI and NVIDIA architectures also outperform the DA layout. We believe that the ASTA layout provides better locality and reduces potential bank conflicts that are more severe on ATI architectures, as current ATI GPUs have simpler DRAM interleaving scheme [2]. Also, when dynamic marshaling is employed, there is an additional marshaing cost for conversion of AoS to DA. This will be addressed in next sections.

For the SpMV benchmark, again both the tiled layout and fully-transposed layout can effectively improve the performance. On ATI architectures, the tiled layouts in general are even faster than the full transpose kernel. We also attribute this effect to shorter strides in the tiled transposition layout.

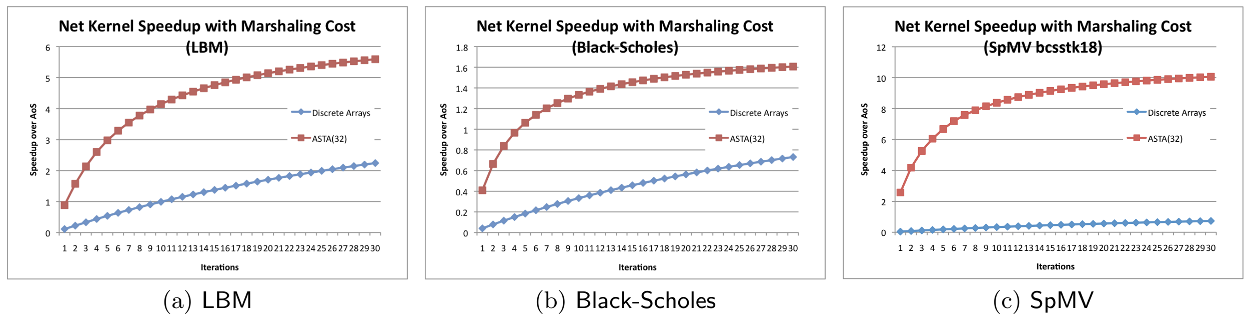
For blackscholes, moderate speedup is obtained on NVIDIA. On ATI DA is slightly faster than AOS and ASTA. That is because the structure size is smaller compared to other applications: only 5 floats. And according to our microbenchmark results earlier, for small structure sizes, AOS is even faster. The reason why DA has speedup on ATI is that out of 5 elements, two are used to store outputs. So DA may create smaller cache footprint as for outputs, other input elements need not to be brought into cache on ATI architecture.

6.1.2 *Performance of Layouts with Marshaling Costs*



(a) NVIDIA (b) ATI

Figure 5: Application Speedup



(a) LBM (b) Black-Scholes (c) SpMV

Figure 6: Net Speedup Including Marshaling Cost

To further understand the cost of layout conversion, or marshaling, Figure 6 shows how the overall speedup including marshaling. Note the cost of marshaling is amortized as the number of iterations increases. The blue curves of all subfigures (DA) are constantly below the red one (ASTA), showing that much more iterations are required to amortize the cost of AoS to DA conversion, and in some cases the net speedup of AoS to DA layout conversion is even below 1.0 given 30 iterations. Whereas AoS to ASTA gives much better overall speedup and break-even point: at most 4 iterations are required to break-even the marshaling cost. Although DA and ASTA have generally comparable performance, clearly the AoS to ASTA layout conversion is much faster than AoS to ASTA then to DA conversion, especially if frequent dynamic layout conversion is required.

6.2 In-place and Out-of-place Marshaling

The ASTA layout, as well as the generalized tiled transposition for tall arrays enable in-place marshaling on GPUs.

To evaluate its performance, we compare an implementation of a highly optimized out-of-place GPU matrix transposition method proposed by Ruetsch and Micikevicius [15] with our in-place tiled transposition kernel.

Since the operation of marshaling does not involve any computation but only memory loads followed by stores, it is sufficient to compare the memory throughput of these two kernels. Table 2 shows the measurements using the CUDA Compute Profiler on an NVIDIA GTX480 GPU on the `e40r0000a` dataset: a 17281 by 17281 sparse matrix stored in ELL format with at most 64 non-zero columns per row.

Table 2: Performance of Full and Tiled Transposition Kernels

Marshaling Kernel (ASTA tile size=16)	Sustained Global Memory Bandwidth (in GB/s)
AoS to SoA [15] (out-of-place)	80.06
AoS to ASTA Barrier-sync (in-place)	82.23
AoS to ASTA PTTWAC (in-place)	19.64

Both the out-of-place kernel and in-place barrier-sync-based kernel utilize local memory to gain coalesced global memory accesses, which still seem to be important for these memory-intensive kernels. On the other hand, cycle-following transposition algorithms naturally suffer from load-imbalance and poor locality. Our PTTWAC algorithm partly addresses the load-imbalance by using atomic operations on parallelized shifts inside cycles, and the use of ASTA layout confines the randomness of memory reference pattern inside a tile, which usually means a handful of cache lines. However, the implementation still suffers from uncoalesced accesses as well as unnecessary contentions caused by simulating bit-wise atomic operations on current GPU architectures. Our PTTWAC-based AoS to ASTA implementation uses atomic operations on local memory to reduce the cost,

and use atomic bitwise operations (AND and OR) to reduce the amount of memory requirement for storing flags in local memory. These contributes to its lower performance. In general, the AoS-to-ASTA PTTWAC algorithm should be consider as an enabler on transposing larger ASTA tiles that beyond the capability of barrier-synchronization-based implementation, rather than a general solution that can replace all other marshaling implementations.

The performance of SoA to ASTA marshaling naturally depends on the number of cycles and cycle length, which are decided by the array size and tile size. We thus compared the performance of two SoA to ASTA marshaling approaches: parallelized IPT (P-IPT) and our algorithm PTTWAC on converting various sparse matrices stored in transposed ELL format into tiled transposed ELL format. These two can be considered as generalized SoA and ASTA layouts.

The performance of P-IPT varies drastically over different input dimensionality as well as ASTA tile sizes, as shown in Figure 7. Across all inputs, PTTWAC performs smoothly and the only significant factor that affects its performance is the tile size. For tile size 64, the performance varies from 15.0 GB/s to 17.40 GB/s, and then performance drops as tile size reduces. This means the imbalance between cycle lengths does not manifest on PTTWAC. However, the P-IPT algorithm, which only parallelizes across cycles, shows unstable performance across inputs of the same tile sizes by almost 5X from 13.65GB/s (bcsstk18, tile size 64) to 3.33GB/s (e40r000a, tile size 64). This matches our prediction that PTTWAC should able to dynamically balance the load by allowing multiple work-groups works concurrently on long cycles.

Table 3 shows the performance of in-place AoS to ASTA transposition, comparing both the approach that use barrier synchronization (BS) and the PTTWAC version. Note for larger tile sizes the BS approach does not work, but when it works, the performance is very good.

Table 3: Performance of In-place AoS to ASTA Transposition (GB/s)

	PTTWAC			BS		
Problem	T=64	T=32	T=16	T=64	T=32	T=16
bcsstk18	8.6	17.0	20.3	55.6	59.4	73.4
e40r000a	6.9	14.0	19.6	51.2	61.0	82.2
bcsstk31	5.3	5.8	8.2	NA	23.2	79.7
bcsstk32	5.0	5.6	7.4	NA	23.8	80.6
s3dkq4m2	7.1	16.6	21.3	61.3	67.0	93.1
conf6.0-	11.7	19.2	20.6	67.9	67.9	86.8

7. RELATED WORKS

In-place transposition on CPU has been well studied by many authors. Readers are encouraged to read Karlsson [13]'s description. He also observed that full in-place transposition can be decomposed efficiently into tiled transposition and local transposition, thus achieving good memory locality on CPUs. For parallelization of in-place matrix transposition, Gustavson & Swirszcz [9] proposed an OpenMP-based approach for multicores that balances the load of each thread by greedily assign cycles to threads, using statically computed cycle leader set from factorization of the matrix dimensions. They mentioned approaches to further break long cycles statically based on a priori knowledge of the cycle lengh. In our approach, no pre-computation and knowledge of the leads and lengths of cycles are required.

On the GPU, the out-of-place transposition problem has been discussed by Ruetsch et al. [15]. And FFT works [7, 8] on GPU also include transposition, but they also seem to be out-of-place algorithms. To our knowledge this work is the first to address transposition of rectangular arrays on GPU.

Jang et al. [12] proposed a methodology for changing the data layout to improve memory coalescing. Zhang et al. [17] proposed a dynamic approach to eliminate irregularities in GPU kernels. The Dymaxion framework proposed Che et al. [5] is a library-based approach that perform marshaling on the CPU side and overlaps PCI-e transfer with the CPU-side marshaling. Both our approach and theirs changes the layout through redefining the mapping function that flatten multidimensional indices into an offset the layout. However, naturally their marshaling performance is limited by the small CPU memory bandwidth and they only allow marshaling between CPU and GPUs, not among different GPU kernels. Also, their approach is equivalent of transforming AoS to SoA, which only improves the memory coalescing but may introduce partition camping as we observed.

In terms of tiling the data structure for memory parallelism, the methodology proposed by Sung et al. [16] is closely related to our approach. They however only transform the kernel and expect manual changes on the host side to reflect the changes in data layout.

On optimizing sparse matrix-vector multiplication, Choi [6] presented manually-optimized sparse matrix layouts to accelerate SpMV for GPUs. However, only the construction, not the conversion between these formats are addressed.

8. CONCLUSION

In this paper, we proposed the Array-of-Structure-of-Tiled-Array (ASTA) layout as a promising alternative to common discrete array transformation for improving the global memory throughput for GPU applications that access data in Array-of-Structure layout. ASTA not only provides better performance to discrete arrays but also enables in-place marshaling on GPUs, which is crucial for accelerators relying on high-throughput access to capacity-constrained private DRAMs. We also show that ASTA allows much faster dynamic in-place marshaling from AoS compared to discrete arrays, which implies much lower break-even point in amortizing the marshaling cost compare to discrete arrays.

We then generalize the ASTA to tiled transposed layouts for arrays that has imbalanced aspect ratios, which is common for sparse matrices. We show that for sparse-matrix transposition such layout also provides comparable or even better performance for a fully transposed layout on sparse matrix-vector kernels.

To allow developers to leverage the benefits of ASTA with minimal effort, this work addresses the problem of decoupling host and device layout needs through a user-friendly automatic transformation framework that is designed and implemented in a transparent way to host code, even allowing user to keep host code unchanged while enjoying the benefit provided by the system.

Acknowledgements

The work is partly supported by the FCRP Giga Scale Research Center, the DoE Vancouver Project

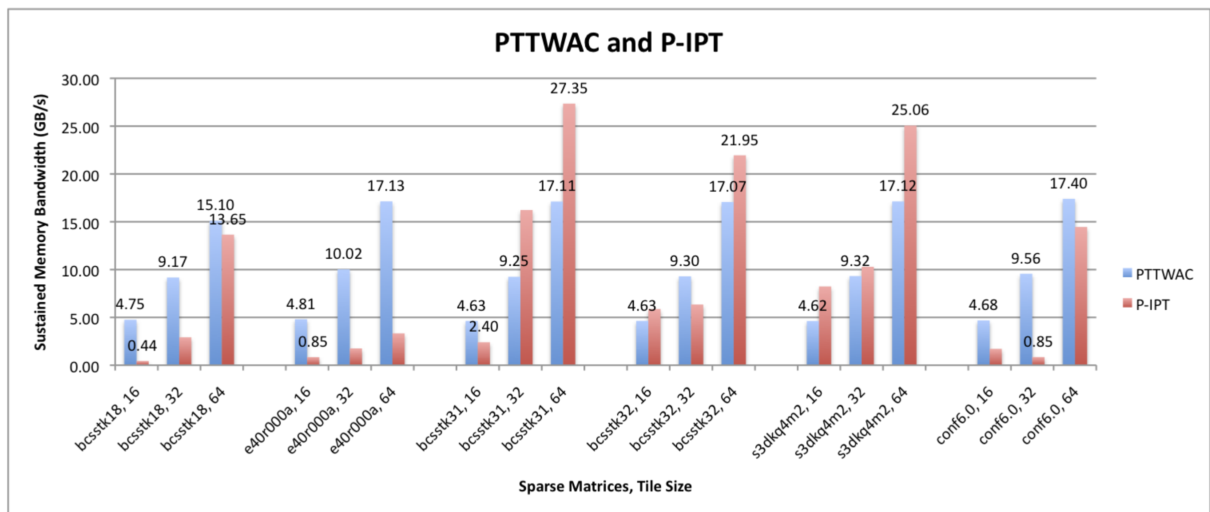


Figure 7: Converting Layout of Array f from SOA to ASTA

(DE-FC02-10ER26004/DE-SC0005515), and the UIUC CUDA Center of Excellence.

9. REFERENCES

[1] ITPACK 2.0 User's Guide. Technical Report CNA-150, Center for Numerical Analysis, Universtiy of Texas, August 1979.

[2] *ATI Stream SDK OpenCL Programming Guide*, June 2010.

[3] The OpenCL Speficication. 2010.

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

[5] 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 (SC)*, Nov 2011.

[6] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, New York, NY, USA, 2010. ACM.

[7] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 257–266, New York, NY, USA, 2011. ACM.

[8] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[9] F. Gustavson, L. Karlsson, and B. Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software*.

[10] F. G. Gustavson and T. Swirszcz. In-place transposition of rectangular matrices. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, pages 560–569, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] T. Hungerford. *Abstract algebra: an introduction.* Saunders College Publishing, 1997.

[12] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):105 –118, jan. 2011.

[13] L. Karlsson. Blocked in-place transposition with application to storage format conversion. Technical report, 2009.

[14] N. R. Mahapatra and B. Venkatrao. The processor-memory bottleneck: problems and solutions. *Crossroads*, 5, April 1999.

[15] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in CUDA. January 2009.

[16] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 513–522, New York, NY, USA, 2010. ACM.

[17] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. *SIGARCH Comput. Archit. News*, 39:369–380, March 2011.