Static Cost Estimation for Data Layout Selection on GPUs

Yuhan Peng
Department of Computer Science
Rice University
Houston, Texas 77005
Email: yp10@rice.edu

Max Grossman
Department of Computer Science
Rice University
Houston, Texas 77005
Email: jmg3@rice.edu

Vivek Sarkar
Department of Computer Science
Rice University
Houston, Texas 77005
Email: vsarkar@rice.edu

***Abstract*—Performance modeling provides mathematical models and quantitative analysis for designing and optimizing computer systems. In high performance architectures, high-latency memory accesses often dominate execution time in many classes of applications. Thus, performance modeling for memory accesses of high performance architectures has been an important research topic. In high performance computation, data layout can significantly affect the efficiency of memory access operations. In recent years, the problem of data layout selection has been well studied on various parallel CPU and some GPU architectures. GPUs have memory hierarchies different from multi-core CPUs. While data layout selection on GPUs has been inspected by several existing projects, there is still a lack of a mathematical cost model for data layout selection on GPUs. This motivates us to investigate static cost analysis methods that could better guide future data layout selection work, and perhaps even designing new SIMT architectures.**

In this paper, we propose a comprehensive cost analysis for data layout selection for GPUs. We build our cost function based on the knowledge of the GPU memory hierarchy, and develop an algorithm which allows researchers to perform compile time cost estimation for a given data layout. Furthermore, we introduce a new vector based representation to represent the estimated cost, which can better estimate the cost of applications with dynamic length loops. We apply our cost analysis to selected benchmarks from past publications on data layout selection. Our experimental results show that our cost analysis can accurately predict the relative costs of different data layouts. Using the cost model presented in this paper, we are developing an automatic data layout selection tool in our ongoing work.

***Index Terms*—GPU, data layout selection, cost estimation, cost vector**

I. Introduction

Currently, the GPU plays an important role in the world of parallel computing. Compared to multi-core CPUs, GPUs have a larger number of computational cores available and are able to better handle heavy computational tasks. Unlike on the CPU, memory transactions are performed at the granularity of warps on the GPU, where every warp consists of a set of adjacent threads. If the threads inside a warp are accessing nearby memory locations, then the memory requests from multiple threads may be satisfiable by a single memory transaction. The technique of combining memory accesses in adjacent threads into fewer memory transactions is called memory coalescing [1]. Thus, compared to the CPU, a poor memory access pattern can lead to greater performance loss on the GPU.

The data layout of an application refers to the manner in which data is stored and organized. Due to the high global memory access latency relative to the latency of arithmetic operations, the choice of data layout can significantly affect the efficiency of execution on the GPU. Moreover, as a result of the unique architectural features of the GPU, the best data layout on the GPU is usually different from the best data layout on the CPU. For instance, in old GPU models before the L1 and L2 cache were introduced, Struct-of-Array (SoA) would usually be preferred since it would achieve better coalescing of memory access. In contrast, on the CPU, using Array-of-Struct (AoS) would be more efficient in general, because using AoS would encourage spatial locality. However, modern GPU models have enabled on-chip L1 cache and off-chip L2 cache. For example, the Fermi GPU has 64KB configurable shared memory and L1 cache, as well as 768KB unified L2 cache [2]. Later models like Kepler and Maxwell further enlarge the size of the L1 and L2 cache [3] [4]. The GPU L1 and L2 cache brought back the opportunity of cache reuse, which means merging fields stored in discrete arrays in SoA into a single AoS might lead to better performance. As a result, the best data layout might be represented in the form of Struct-of-Array-of-Struct (SoAoS), a hybridization of AoS and SoA.

Determining the best data layout for a GPU kernel is difficult because the number of choices of different data layouts can be numerous. For example, suppose there are N fields, then the number of different SoAoS layouts is the Nth Bell number B(N), where B(20) is already over 50 trillion [5]. Furthermore, finding the best data layout has been proved to be NP-hard [6] [7]. In recent years, automatic data layout selection had been studied on GPUs [8] [9] [10]. These approaches clearly identified the benefit of using SoA for memory coalescing as well as using AoS for cache reuse, and they propose polynomial time algorithms to approximate the best data layout. Unfortunately, none of these approaches proposed a mathematical cost model to estimate the cost for a given data layout. Without a mathematical cost model, the performance of the data layout selection algorithm can be only examined from

the performance results of selected benchmarks, which might be misleading. This motivates us to build a unified cost model that can be further applied to all data layout selection algorithms.

In this project, we introduce a comprehensive mathematical cost model based on the GPU memory hierarchy, as well as an algorithm to estimate the cost of a given data layout at compile time. We also propose a novel cost vector representation to better handle cost estimation in the presence of dynamic length loops. Our cost model is not limited to SoAoS, and can be applied to any data layout selection problem on GPUs. Our goal is to build an accurate cost model for data layout selection on GPUs, where data layouts with lower costs will have better execution efficiency.

The remainder of this paper is organized as follows: Section II provides background on the basic terminologies and concepts of the GPU computation model and data layout to aid in understanding our approach for the cost analysis, as well as our motivation for building a comprehensive cost model for data layout selection on GPUs. Section III introduces the design of our cost function. Section IV describes our algorithm for static cost estimation based on the GPU kernel and a given data layout. In Section V we apply our cost estimation algorithm on selected benchmarks, and compare our estimated cost with the actual performance result. Section VI discusses related works on GPU performance evaluation and data layout selection. Section VII summarizes our conclusions and the ongoing extension of this project.

II. Background

A. GPU Global Memory Access Mechanism

In order to improve global memory transaction efficiency, modern GPU models have added both L1 and L2 caches. The on-chip L1 cache has a relatively smaller size, but also a lower latency and higher bandwidth. The L1 cache is shared by all threads inside the same Streaming Multiprocessor (SM), where there might be multiple CUDA thread blocks inside each SM. For Fermi there can be up to 8 active CUDA thread blocks per SM, and for Kepler there can be up to 16 active CUDA thread blocks per SM [3] [11]. The L1 cache shares the same on-chip memory with CUDA shared memory, and the user can partition the memory space between the L1 cache and shared memory. In contrast, the L2 cache has a larger size, but a much higher latency. The L2 cache is shared by all threads inside the GPU.

The scheduling of threads on the GPU is done at the granularity of warps. On current Nvidia GPUs, every 32 consecutive threads form a warp. Threads inside a warp share the same program counter, but have their own registers and local memory space [12]. At each cycle, all threads inside the warp may execute one instruction in a single instruction, multiple thread (SIMT) manner. If there are branches, all threads will execute all possible branch paths. For each branch path, threads not going through this path will be disabled. After all paths are taken, threads will be resumed for further execution. The execution of different warps is independent of each other.

Global memory accesses are done in 32, 64 or 128-byte transactions aligned at 32, 64, or 128-byte addresses. When accessing global memory, the warp determines the transaction length according to the type of the access (read or write) and the cache hierarchy of the chip. On some GPU models such as Kepler, a 128-byte transaction will be reduced to a 64-byte transaction if only half of the 128 bytes is actually needed, and a 64-byte transaction can be further reduced to a 32-byte transaction [13]. The number of actual transaction requests will be equal to the number of unique memory segments. For example, if all threads in the warp are accessing the same memory location, then only one global memory transaction will be needed. If all threads inside a 32-thread warp are accessing adjacent 4-byte memory locations, and the memory location accessed by the first thread is a multiple of 128 bytes, then only a single contiguous 128-byte memory segment will be accessed, and this is an example of a fully coalesced memory access.

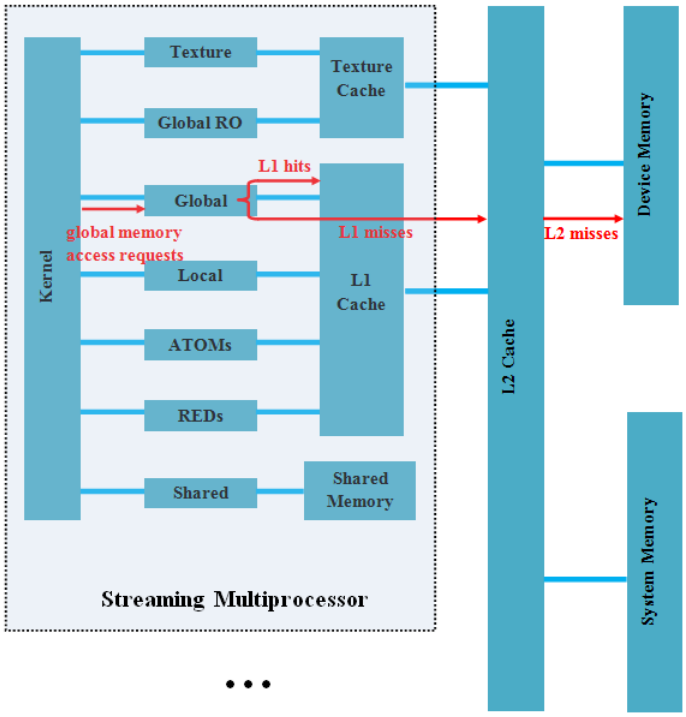

Fig. 1. The hierarchy of GPU memory, modified from the page [14]. From the figure, we can see the L1 cache are shared by all the threads inside the same SM, and all the SMs are sharing one L2 cache. The red lines indicate the path of physical memory locations where global memory access requests could traverse.

Like on the CPU, global memory transactions may go through the L1 and L2 cache. If the L1 cache is present and enabled, then global loads will first try to hit in the L1 cache. If a load transaction missed in the L1 cache, it will then try to hit in the L2 cache. For current GPUs, global stores are not cached in the L1 cache, as they directly go to the L2 cache. Similarly, the L2 cache will be accessed if

the memory segment is found in the L2 cache, and GPU Dynamic Random-Access Memory (DRAM) will be accessed if the transaction misses in the L2 cache. Memory segments will be cached in both L1 and L2 cache when being loaded from the DRAM.

Figure 1 shows an example of the GPU memory hierarchy, together with the physical memory locations a global memory access request might traverse. For this project, we are only considering the benefit of the L1 and L2 cache. However, from the figure, we can still see GPU has many special-purpose memories such as shared memory and texture memory. We will study the modeling of these special-purpose memories in our future work.

B. The Impact of Different Data Layouts

The selection of data layout can highly impact performance on both CPU and GPU, as different data layouts might influence the cache hit ratio, as well as the extent of memory access coalescing. The most common choices for data layout are Array-of-Struct (AoS) and Struct-of-Array (SoA).

In AoS, all fields for a logical data point are grouped into the same struct, and one array is declared for the grouped struct. The following code shows a sample AoS declaration.

```
typedef struct
{
   int x;
   int y;
   int z;
} AoS;
AoS arr[N];
```

In SoA, discrete arrays are declared for different fields. The following code shows a sample SoA declaration.

```
typedef struct
{
   int x[N];
   int y[N];
   int z[N];
} SoA;
SoA arr;
```

The main difference between AoS and SoA is how the data is organized in memory. In AoS, fields for the same logical data point stay together. In SoA, the same fields tend to converge while different fields tend to diverge. Past work has demonstrated that for many applications, the AoS layout can improve CPU cache hierarchy utilization, because different fields for the same logical data point are often accessed together, and in AoS different fields with same array index are declared in the same structure. Accessing one field will also bring the adjacent fields into the cache. On the other hand, SoA may not utilize the cache well, as accessing `x[i]` will not result in loading `y[i]` into the cache since `x[i]` and `y[i]` are unlikely to be in the same cache line. Therefore, AoS is usually preferred on the CPU.

However, on the GPU, using SoA will be more likely to result in more coalesced memory accesses, because when all threads in the same warp are accessing the same field for different logical data points simultaneously, using SoA will lead to fewer memory transactions as the same fields are declared in adjacent memory locations. In contrast, AoS might lead to worse coalesced accesses, as declaring different fields in the same struct will waste memory bandwidth when accessing only a single field. Thus, SoA was preferred on the GPU before the GPU cache was introduced.

With the introduction of the GPU's cache hierarchy, merging closely accessed fields into the same structure may bring back the benefit of cache reuse. As a result, the best data layout may be some variant of Struct-of-Array-of-Struct (SoAoS). The following example shows a sample SoAoS declaration, designed for the scenario when fields `x` and `y` are accessed closely, and field `z` is accessed away from fields `x` and `y`.

```
typedef struct
{
   int x;
   int y;
} AoS;
typedef struct
{
   AoS InnerArr[N];
   int z[N];
} SoAoS;
SoAoS arr;
```

Figure 2 shows an example of the memory organization of the AoS, SoA and SoAoS declarations above, with N = 100.

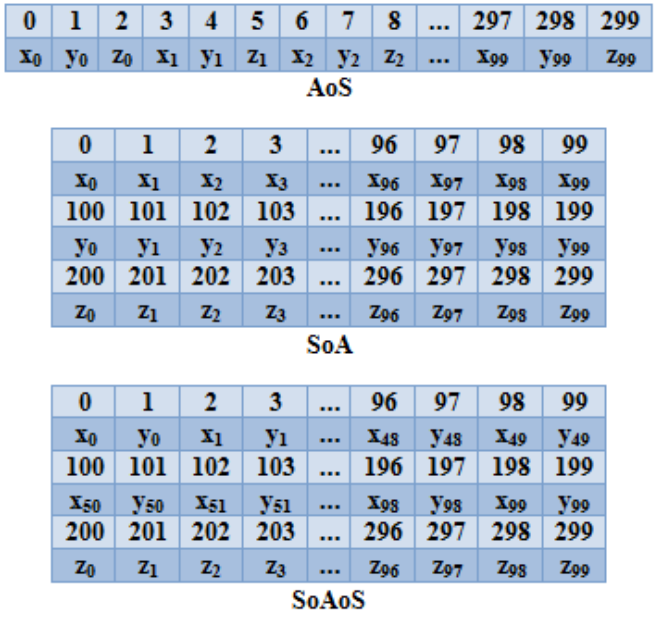

Fig. 2. The memory organization of our sample AoS, SoA, and SoAoS with N = 100. It can be seen in AoS, x[i], y[i] and z[i] are stayed together for all i from 0 to 99. In SoA, all field x are staying before all field y, and all field y are straying before all field z. In SoAoS, the field x and y are organized like in AoS, while all field x and y are staying before all field z.

The number of possible SoAoS is the Bell number of the number of fields, which grows exponentially. Moreover, finding the optimal SoAoS is proven to be NP-hard. Most of the existing approaches use a distance-based algorithm for automatic data layout selection. These approaches start from SoA, and merge fields which are accessed within a predefined instruction distance. More details about existing distance-based approaches will be discussed in Section VI-B. However, none of these approaches proposed a mathematical cost function to minimize, which can make objective analysis of these approaches difficult. This motivates us to develop a comprehensive cost analysis mechanism for data layout selection on GPUs.

In this project, we first propose a cost function based on data layout and kernel memory accesses. Then we provide an algorithm that enables the cost estimation for a given data layout at compile time. Our goal is to develop an accurate cost function and static cost estimation algorithm, so that the data layouts with lower estimated cost will have smaller cost function value and less running time.

III. Cost Function

In this section, we introduce our cost function for data layouts. In order to emulate the real cost, we build our cost function based on knowledge of the GPU memory hierarchy. At a high level, the cost function multiplies the number of global transactions that are likely to target L1, L2, and DRAM with the overhead of accessing those memory hierarchy levels and then sums up the product terms.

To build up our cost function, we first define the cost coefficient as the overhead of accessing different memory hierarchies. We define W_{L1}, W_{L2} and W_{DRAM} as the costs for accessing the L1 cache, accessing the L2 cache, and accessing the DRAM, respectively. Since off-chip memory accesses take significantly more time than on-chip memory accesses, W_{L1} is usually much smaller than W_{L2} and W_{DRAM}. Furthermore, hitting in the L2 cache is still preferred relative to accessing the DRAM.

Then we use N_{read} and N_{write} to denote the total number of global memory transaction requests made during the execution of the kernel. Moreover, we define R_{L1_read}, R_{L2_read} and R_{L2_write} as the overall L1 hit ratio for read requests, the overall L2 hit ratio for read requests, and the overall L2 hit ratio for write requests. Then we could calculate the cost according to the formula below.

$$Cost_{read} = W_{L1} * N_{read} * R_{L1\_read} + W_{L2} * N_{read} * (1 - R_{L1\_read}) * R_{L2\_read} + W_{DRAM} * N_{read} * (1 - R_{L1\_read}) * (1 - R_{L2\_read})$$

$$Cost_{write} = W_{L2} * N_{write} * R_{L2\_write} + W_{DRAM} * N_{write} * (1 - R_{L2\_write})$$

$$Total\ Cost = Cost_{read} + Cost_{write}$$

All the variables in the above formula can be retrieved from GPU profiling metrics when executing the kernel. This offers a quantitative and precise target to validating our cost estimation algorithm described in Section IV. The goal of data layout selection is to find out the data layout which minimizes the total cost.

IV. Cost Estimation

To estimate the cost from a given data layout at compile time, we have to estimate the cost of different memory access operations separately and sums up all costs. To compute the cost of a memory access operation, we have to estimate the following terms for that operation, and we use the ~ symbol to refer to the estimated values:

- Cost coefficient: $\widetilde{W}_{L1}$, $\widetilde{W}_{L2}$, and $\widetilde{W}_{DRAM}$.
- Number of transaction requests: $\widetilde{N}_{read}$, $\widetilde{N}_{write}$.
- Cache hit ratio: $\widetilde{R}_{L1\_read}$, $\widetilde{R}_{L2\_read}$ and $\widetilde{R}_{L2\_write}$.

A. Assumptions

It is difficult to estimate the behavior of the GPU cache at compile time because both L1 and L2 caches are shared by many warps, and the scheduling of warps is not predictable at compile time. Moreover, our cost model is not intended to estimate the actual running time, but rather to determine whether a given data layout is better than another in terms of the total cost of global memory accesses. Thus, we make the following assumptions.

1) Both L1 and L2 cache adopt LRU cache replacement policy, as most modern GPUs use LRU or pseudo-LRU for both L1 and L2 caches [15].
2) We assume all threads are running in lockstep as a large, cross-SM warp, i.e. they are executing the same instruction concurrently, before moving on to the next instruction. The reason for making this assumption is that it is hard to determine the order of warp execution at compile time. Although this assumption may overestimate the cache hit ratio, it is one of the most reasonable ways to estimate cache behavior at compile time.
3) Moreover, we only consider the L1 and L2 cache for the memory locality. On current GPUs, the shared memory and the L1 cache are sharing the same piece of on-chip memory, hence have the same latency. The only difference is that the shared memory is utilized explicitly, and the L1 cache is utilized implicitly. Thus, under our cost model, we can treat the shared memory like the L1 cache. More detailed studies about shared memory and other on-chip special-purpose memories are left for future work.

B. Basic Terms of Accessing Global Memory

Before describing our cost estimation algorithm, we first define some basic terms about accessing global memory. Although some of these terms may have been mentioned in previous studies on data layout selection, we provide our own definitions to ensure our estimation algorithm is unambiguous. The terms in this section are not limited to SoAoS selection, although we will illustrate these terms using SoAoS.

In general, each global array `Arr` in the SoAoS layout can be declared in the following format.

```
typedef struct {
   type x;
   type y;
   type z;
   ...
} StructType;
StructType Arr[Len];
```

The `type` here can be any 1, 2, 4 or 8-byte data type. There will be some padding bytes to ensure the starting address alignment of every N-byte field is a multiple of N bytes. `StructType` may have one or more fields inside.

Then for each access to the field `x` inside the array `Arr`, where `Arr` is an array of `StructType`, we define the following terms.

1) Block Size, Grid Size, Number of Blocks per SM, and Thread ID: For a GPU kernel, the *block size* is the number of threads inside each CUDA thread block, and the *grid size* is the number of CUDA thread blocks in total. The *number of blocks per SM* is the maximum number of CUDA thread blocks each SM can have when executing the kernel, and it is bounded by the GPU architecture as well as the amount of resources each block is requesting. For each thread, the *thread ID* (tid) is its one-dimensional rank inside the global thread pool.

2) Structure Size: The *structure size* of a structure is the number of bytes of the structure including the padding bytes. For example, the structure size of the following `MyType` structure is (1 + 3 + 4 + 1 + 1 + 2) = 12 bytes, due to the padding bytes after fields `w` and `y`.

```
typedef struct {
  char w; // 1 byte
    // 3 padding bytes after w to ensure the
    //  starting address of y is multiple of 4
  int x; // 4 bytes
  char y; // 1 byte
    // 1 padding byte after y to ensure the
    //  starting address of z is multiple of 2
  short z; // 2 bytes
} MyType;
```

3) Array Index: The *array index* for accessing data field `x` using AoS as `Arr[i].x`, or using SoA as `Arr.x[i]`, will both be `i`. Array indices can also be mathematical expressions or even functions of variables. For example, `Arr[i*j].x` and `Arr[foo(i+j)].x` are allowed in the cost estimation, although we conservatively handle these cases.

4) Stride: For a global access to data field `x`, the *stride* between a pair of adjacent threads in the same warp is the difference between the two memory addresses of the field `x` these two threads are accessing. The value of the stride depends on the difference of the array indices accessed between adjacent threads, as well as the structure size of the structure field `x` belongs to. Strides can be different for different pairs of adjacent threads.

The following formula shows how to calculate the stride between thread `tid` and `tid + 1`.

```
indexDiff = abs(array_index(tid) - array_index
    (tid + 1))
stride(tid, tid + 1) = indexDiff * sizeof(
    StructType)
```

Sometimes, `indexDiff` cannot be computed at compile time, especially when array indices are non-analyzable functions of the thread ID. In such cases, we would say the stride cannot be computed at compile time, or the stride is undefined for static cost estimation.

Below is an example of vector addition using AoS and SoA layout. Consider the following sample code written in CUDA.

```
__global__ void vectorAdditionAoS (MyAoS *
    inputStruct, float *result)
{
   const int tid = blockIdx.x * blockDim.x +
       threadIdx.x;
   float tempX = inputStruct[tid].x;
   float tempY = inputStruct[tid].y;
   result[tid] = tempX + tempY;
}

__global__ void vectorAdditionSoA (MySoA *
    inputStruct, float *result)
{
   const int tid = blockIdx.x * blockDim.x +
       threadIdx.x;
   float tempX = inputStruct->x[tid];
   float tempY = inputStruct->y[tid];
   result[tid] = tempX + tempY;
}
```

We can see if using AoS, both accesses to `x` and `y` has array_index(`tid`) = `tid`, thus `indexDiff` will be 1 for all adjacent threads. Thus all strides will be `indexDiff` * sizeof (`MyAoS`) = 8 bytes. On the other hand, if using SoA, both accesses to `x` and `y` also has array_index(`tid`) = `tid`, then `indexDiff` will also be 1 for all adjacent threads. Thus all strides will be `indexDiff` * sizeof (int) = 4 bytes.

5) Instruction Distance: We define the *L1 and L2 instruction distance* between two memory access instructions I_1 and I_2, as the number of unique memory locations accessed by all threads sharing the L1 or L2 cache between I_1 and I_2. According to our second assumption in Section IV-A, different threads will have the same contribution to the instruction distance. Moreover, fetching a field into the cache will also fetch the fields declared in the structure if they fall in the same cache line, and these fields will also contribute to the unique memory locations even if they were not accessed.

Then, the L1 instruction distance between two memory access instructions can be calculated as:

```
#_blocks_per_SM * block_size * unique memory
    locations in between (inclusively) in one
    thread
```

And the L2 instruction distance between two memory access instructions can be calculated as:

```
grid_size * block_size * unique memory
    locations in between (inclusively) in one
    thread
```

Based on our LRU assumption in Section IV-A, there will be no benefit of cache reuse between accesses with instruction distance larger than the cache size.

For example, for the following accesses to `Arr1[i].x` and `Arr1[i].y`, where `Arr1` is an array of structure containing integer fields `x` and `y`, and `Arr2` is an array of structure containing integer fields `w` and `z`. All fields are 4-byte data type.

```
... = Arr1[i].x // Instruction A
... = Arr2[i].z
... = Arr2[i].w
... = Arr2[i].z
... = Arr2[i].y // Instruction B
```

Suppose the grid size is 1024, the block size is 256, and the number of blocks per SM is 8. Then between Instruction A and B, the L1 instruction distance is 8 * 256 * 4 * 4 = 32768 bytes, and the L2 instruction distance is 1024 * 256 * 4 * 4 = 4194304 bytes.

6) Index Accordance: In a thread, for accesses to different fields `x` and `y` which are candidates for cache reuse, we are less interested in the actual difference between their array indices. Instead, we are more interested if these two instances of field `x` and `y` will be in the same cache line. Thus, we say two accesses are *L1_beneficial* if the elements accessed can be proved to be in the same L1 cache line, and they are *L2_beneficial* if the elements accessed can be proved to be in the same L2 cache line. In our approach, index accordance is only meaningful for fields declared in the same structure.

For example, the following accesses to x and y are neither L1_beneficial nor L2_beneficial, as we cannot guarantee that `Arr1[i].x` and `Arr1[N - i].y` will be in the same L1 or L2 cache line.

```
... = Arr1[i].x
... = Arr1[N - i].y
```

C. Estimating the Cost Coefficient

Because the GPU has the ability to overlap memory accesses with computational instructions, precisely defining the cost of different types of memory accesses can be difficult. However, since arithmetic operation latency on the GPU usually consumes significantly fewer cycles than memory access operations, a conservative approach is to assign cost coefficients based on memory access latency normalized to a hit in the L1 cache. Unfortunately, there is no official release of latencies of the GPU memory hierarchy. Table I shows some approximated latency values for Fermi from running profiling benchmarks [16] [17].

Thus, we would set $\widetilde{W}_{L1}$= 1, $\widetilde{W}_{L2}$ = 30, $\widetilde{W}_{DRAM}$ = 100 for Fermi as relative costs. For difference GPU architectures, we will set $\widetilde{W}_{L1}$, $\widetilde{W}_{L2}$ and $\widetilde{W}_{DRAM}$ in a similar way.

Memory Hierarchy	Latency (Cycles)
L1 Cache	10
L2 Cache	300
DRAM	1000

TABLE I
LATENCIES FOR FERMI MEMORY HIERARCHY.

D. Estimating the Number of Transaction Requests

Stride is the only attribute needed to estimate the number of transaction requests. For each memory access instruction, there are three common cases:

- If all strides within a warp equal 4 bytes, we say the access has a *unit stride*, or the access is *fully coalesced*. In this case, if the memory address of the field to be requested from the first thread is a multiple of 128 bytes, then inside a warp, there will be only one global memory transaction. Moreover, if all strides = 0 inside a warp, i.e. all threads are accessing the same element, then only one global memory transaction will be needed for all threads inside the same warp as well.
- If all strides equal to N * 4 inside a warp, for some N where 1 < N < 32, then inside a warp there will be N 128-byte transactions needed.
- If all strides can be proven to be larger than or equal to 128 bytes, where they do not have to be equal among different warps, then we say the access is *fully non-coalesced*. In such case, inside a warp, 32 separate global memory transactions will be needed. Moreover, if the array index is independent of the thread id, and the stride cannot be analyzed at compile time, then we will regard the access as fully non-coalesced.

Unfortunately, in some cases, the stride cannot be known at compile time since the array indices may be functions of runtime variables. In these cases, we make the conservative estimation that all dynamic strides are larger than 128 bytes.

Finally, inside a warp, by examining all the strides as well as all the relative memory address, we would be able to determine the number of unique memory segments requested by the warp.

E. Estimating the Number of Blocks per SM

In order to calculate the L1 instruction distance, we have to firstly estimate the number of blocks per SM. When executing a kernel, the number of blocks per SM is restricted by the following factors:

- The maximum number of thread blocks an SM can have for the GPU architecture, *max_#_blocks_per_SM*.
- The maximum number of threads an SM can have for the GPU architecture, *max_#_threds_per_SM*. For any kernel, we always have

  ```
  block_size * #_blocks_per_SM <= max_#
      _threds_per_SM.
  ```

- The number of registers an SM has for the GPU architecture, *#_registers_per_SM*, which can be retrieved using `cudaGetDeviceProperties` API [18].

Moreover, when compiling a kernel, we can retrieve the number of registers requested by the kernel, *#_registers_per_thread*. For example, we may pass the `-Xptxas="-v"` option if using the `nvcc` compilation command [19]. Finally, we should have

```
#_registers_per_thread * block_size * #
    _blocks_per_SM <= #_registers_per_SM
```

As a summary, we can use the following formula to estimate the number of blocks per SM for a kernel.

```
#_blocks_per_SM = max(1, min(max_#
    _blocks_per_SM, max_#_threds_per_SM /
    block_size, #_registers_per_SM / #
    _registers_per_thread / block_size))
```

F. Estimating the Cache Hit Ratio

We will use the following rule to estimate the cache hit ratio: for an access to field y after an access to field x, if and only if their instruction distance is no greater than the L1 or L2 cache size, and the two accesses are L1_beneficial or L2_beneficial, then y may be a hit in the L1 or L2 cache.

To determine if the index accordance is satisfied, we have to look at the array indices, as well as the structure size. Suppose we have two accesses to Arr[i].x and Arr[j].y. First, if the difference of array indices, i.e. the absolute value of i – j, cannot be known at compile time, then we will say the index accordance is not satisfied, although this may miss some opportunities for cache reuse. Otherwise, we look into such difference times the size of the struct of the array Arr.

```
(|i - j| + 2) * sizeof(dataType(Arr))
```

Then, we say the L1 or L2 index accordance will be satisfied, if and only if the above value is less than or equal to the L1 or L2 cache line size.

An access to the field x will be hit in the L1 or L2 cache if there exists some field y declared in the same structure with x, where there is an access to the field y before the access to the field x within the L1 or L2 instruction distance, and the L1 or L2 index accordance was satisfied. Otherwise, DRAM will be accessed for accessing field x. Among different threads inside the same warp, whether the access will be hit in the L1 or L2 cache may be different. Finally, the cache hit ratio can be retrieved by considering the cache hit or miss of all threads.

G. Estimating the Cost

To estimate the cost, we must have access to the kernel, a given data layout, as well as:

1) Grid size and block size. We also provide default values for them if they are not explicitly specified.
2) The number of threads per warp. For all current Nvidia GPUs this value is 32.
3) L1 cache size. For Fermi the L1 cache size can be configured as 16KB or 48KB, and for Kepler the L1 cache size can be configured as 16KB, 32KB or 48KB. We set this value to be 0 if the L1 cache is not enabled.
4) L1 cache line size. For all current GPUs this value is 128 Bytes.
5) L2 cache size. The L2 cache size for Fermi is 768KB, and for Kepler it is 1536KB.
6) L2 cache line size. For all current GPUs this value is 32 Bytes.

Theoretically, to compute the total amount of memory transactions, we would need to consider each execution of every memory access instruction. However, in practice, we usually cannot retrieve how many times each memory access instruction will be executed at compile time, especially when the instruction is in a loop with dynamic length. Thus we may need to give an estimation of the value of dynamic loop lengths. In practice, our give the estimation of the length of all dynamic loops as 100. Then we can unroll all the loops, and transform the kernel into an extend basic block.

In the transformed code, for each global memory access instruction, we estimate the cost in the granularity of warps. We use the approaches in Section IV-D to estimate the number of global transaction requests, and use the rules in Section IV-F to estimate the cache hit ratio. Then we multiply the number of transactions hit in the L1 cache, hit in the L2 cache, and go to DRAM with their corresponding cost coefficient, which will be estimated using the rules introduced in Section IV-C. Finally, the cost of the memory access instruction can be calculated by summing up the costs from all warps.

Finally, adding up the cost of all global memory accesses, we will get the total estimated cost.

In practice, given two different data layouts, using our cost estimation algorithm we are able to compute the cost of each data layout. The layout with lower cost will potentially have a better memory access efficiency, hence smaller execution time.

H. Handling Dynamic Loop Lengths

Using a constant to approximate the length of dynamic loops is a standard approach in static cost estimation. However, in our experience, this approach can be less effective when some loops have dynamic lengths and others have lengths that are compile-time constants. Thus, as in big-O analysis, where variable terms are assumed to be more significant than constant terms, we take an approach in which loops with variable lengths are considered to be more significant than loops with constant lengths.

We start by defining the *complexity degree*, $CD(I)$, of memory access instruction, I, as the number of loops enclosing I with variable loop length. We also define the *complexity vector*, $CV(I)$, of memory access instruction, I, enclosed in n loops as a vector of size $n+1$ with a 1 at position $CD(I)$ and zeroes in all other positions.

For example, consider the following loop nest in which M and N are unknown at compile time:

```
for (i = 0; i < M; i++)
   for (j = 0; j < 128; j++)
```

```
for (k = 0; k < N; k++)
   {
      ...
      a[index].x ++; // Instruction A
      ...
   }
```

Then the complexity degree of instruction A will be $CD(A) = 2$, since there are two loops with unknown length enclosing A, and the complexity vector will be $CV(A) = (0,0,1,0)$ with a 1 in position 2, thereby representing a quadratic complexity.

During cost estimation, for each memory access instruction, once we have computed its estimated cost, we multiply it by the complexity vector for that instruction. The resulting vector is called the *cost vector*. Two cost vectors can be added using standard vector addition, and we use this way to sum up the total cost.

Finally, we have to be able to determine if one cost vector is larger than the other. Since the rightmost entry of a cost vector is the most significant entry, we say that (X_0, X_1, ..., X_n) > (Y_0, Y_1, ..., Y_n), if and only if, there exists some 0 < m <= n, ($X_m > Y_m$) and ($X_{m+1} = Y_{m+1}$) and ($X_{m+2} = Y_{m+2}$) ... and ($X_n = Y_n$).

Our vector-based representation will prefer data layouts which focus on optimizing accesses within dynamic length loops, which will have a higher chance of being executed more frequently. For example, for the following code, Instruction A and B are the only global memory access instructions, and the value of M and N cannot be known at compile time. Suppose the grid size is 100, the block size is 256, and the number of blocks per SM is 8. Assume that we have two data layouts: Layout_1 and Layout_2. To make the example simple, suppose using Layout_1, for each warp, the cost of every execution of Instruction A is always 1 and the cost of every execution of Instruction B is always 3. Similarly, if using Layout_2, for each warp, the cost of every execution of Instructions A is always 3 and the cost of every execution of Instruction B is always 1.

```
for (i = 0; i < M; i++)
{
   for (j = 0; j < 128; j++)
   {
      ... = a[index1].x; // Instruction A
   }
   for (k = 0; k < N; k++)
   {
      ... = a[index2].y; // Instruction B
   }
}
```

First, we see that the complexity degrees for instructions A and B are $CD(A) = 1$ and $CD(B) = 2$, yielding $CV(A) = (0,1,0)$ and $CV(B) = (0,0,1)$ Then, if we assume the loop i and k will execute 100 times, the cost vector for Layout_1 will be 100 * 128 * 100 * 256 * 1 * (0, 1, 0) + 100 * 100 * 100 * 256 * 3 * (0, 0, 1) = (0, 327680000, 768000000).

And the cost vector for Layout_2 will be 100 * 128 * 100 * 256 * 3 * (0, 1, 0) + 100 * 100 * 100 * 256 * 1 * (0, 0, 1) = (0, 983040000, 256000000).

Since (0, 327680000, 768000000) > (0, 983040000, 256000000), Layout_2, which better accommodate Instruction B, will be preferred by our cost estimation algorithm. The reason is we believe Instruction B will be executed more frequently than Instruction A in practice.

V. Performance Evaluation

The performance evaluation is done in the following way. First, for each benchmark, we propose some reasonable data layouts, including AoS and SoA. Then we compute the cost for each data layout using our cost estimation algorithm. After that, we run the benchmark written in each of the chosen data layouts, and record the total running time and all profiling metrics related to global memory access operations. We can compute the dynamic cost using the profiling metrics and the relative costs $\widetilde{W}_{L1}$, $\widetilde{W}_{L2}$ and $\widetilde{W}_{DRAM}$. Finally, we compare the actual running time with the estimated cost and the dynamic cost. We expect the layout with lower estimated cost will have smaller dynamic cost, as well as smaller execution time.

A. Machine Configuration

We run our benchmarks on both Fermi and Kepler GPUs. The Fermi GPU we used was an Nvidia Tesla M2050 GPU [20]. The Kepler GPU we used was an Nvidia Tesla K20c GPU [21]. In this generation of Kepler GPU, the L1 cache is not enabled for caching global memory transactions. Table II shows more detailed attributes of the machines used during performance evaluation.

Attribute	Tesla M2050	Tesla K20c
# Cuda Cores	448	2496
# Threads per Warp	32	32
Max # Blocks per SM	8	16
Max # Threads per SM	1536	2048
# Registers per SM	32768	65536
Max # Registers per Thread	63	255
L1 Cache Size	64KB	0KB
L1 Cache Line Size	128 Bytes	N/A
L2 Cache Size	768KB	1536KB
L2 Cache Line Size	32 Bytes	32 Bytes

TABLE II
RELATED ATTRIBUTES OF THE TESTING MACHINES.

B. Benchmarks

In this paper, we present results for the following benchmarks. All these benchmarks have their unique feature to be studied in data layout selection problems.

1) K-means: K-means is a clustering algorithm in data mining. It keeps assigning the points to the cluster, recomputes the centroid of the new clusters, until reaching a fixed point. Inside the kernel, although the fields `feature` and `clusters` field are accessed within a short instruction distance, however, the index accordance was not satisfied. Thus using AoS will not be beneficial from the cache reuse. For this benchmark, we choose AoS, SoA, and SoAoS which merges the fields `feature` and `clusters`.

2) N-body: N-body is the physics algorithm that predicts the motion of each object by considering the forces from all other objects. Except loading and storing the coordinates of the current object, all of the other global accesses are accessing the same element at each time. Thus most of the accesses can be done within one transaction in every warp. We choose data layouts AoS, SoA, and two versions of SoAoS. The first SoAoS groups input fields `x`, `y`, `z` and output fields `vx`, `vy` and `vz`, and the second SoAoS groups `x` with `vx`, `y` with `vy`, and `z` with `vz`.

3) LavaMD: LavaMD simulates the relocation of particles in 3D space. The computation is performed on the granularity of cubes, where each cube has 26 surrounding neighbors. Since the grid computation will take the advantage of cache reusing, using AoS for each point might be more beneficial. We choose data layouts AoS, SoA and two versions of SoAoS. Like N-body, the first SoAoS groups all the dimensions of the input point, as well as the output point. The second SoAoS groups each dimension of the input and the output point.

4) Summary: Table III shows the benchmarks and data layouts we selected for our performance evaluation.

Benchmark	Data Layout	Field Grouping
K-means	AoS	{feature, clusters, membership}
K-means	SoA	feature, clusters, membership
K-means	SoAoS	{feature, clusters}, membership
N-body	AoS	{x, y, z, vx, vy, vz}
N-body	SoA	x, y, z, vx, vy, vz
N-body	SoAoS 1	{x, y, z}, {vx, vy, vz}
N-body	SoAoS 2	{x, vx} , {y, vy}, {z, vz}
LavaMD	AoS	{d_rv_gpu.v, d_rv_gpu.x, d_rv_gpu.y, d_rv_gpu.z, d_fv_gpu.v, d_fv_gpu.x, d_fv_gpu.y, d_fv_gpu.z}
LavaMD	SoA	d_rv_gpu.v, d_rv_gpu.x, d_rv_gpu.y, d_rv_gpu.z, d_fv_gpu.v, d_fv_gpu.x, d_fv_gpu.y, d_fv_gpu.z
LavaMD	SoAoS 1	{d_rv_gpu.v, d_rv_gpu.x, d_rv_gpu.y, d_rv_gpu.z}, {d_fv_gpu.v, d_fv_gpu.x, d_fv_gpu.y, d_fv_gpu.z}
LavaMD	SoAoS 2	{d_rv_gpu.v, d_fv_gpu.v}, {d_rv_gpu.x, d_fv_gpu.x}, {d_rv_gpu.y, d_fv_gpu.y}, {d_rv_gpu.z, d_fv_gpu.z}

TABLE III
BENCHMARKS AND DATA LAYOUTS FOR OUR PERFORMANCE EVALUATION

C. Performance Results

The reference code for our N-body benchmark was modified from Nyland's Fast N-body implementation [22]. The reference code for our K-means and LavaMD benchmarks was modified from Rodinia's implementation [23]. All the codes are written in CUDA. To simplify comparison of our results, all the loop lengths are given at compile time so that the estimated cost can be represented by a single value.

Table IV shows the performance results on Fermi, and Table V shows the performance results on Kepler. For each benchmark, the data layouts are sorted by increasing order of the speedup. For the estimated cost and the dynamic cost, since the actual number can be very large, we give the relative ratio against the cost of the AoS layout. Since the cost of memory access operations is only part of the execution overhead, we do not expect the ratio of the cost to be approximately equal to the ratio of the execution time, but they are supposed to have a positive correlation.

Benchmark + Data Layout	Estimated Cost Against AoS	Dynamic Cost Against AoS	Speedup Against AoS
K-means (SoAoS)	0.667	0.663	1.208×
K-means (SoA)	0.333	0.399	1.531×
N-body (SoAoS 2)	1.003	1.272	0.917×
N-body (SoA)	0.826	0.905	1.088×
N-body (SoAoS 1)	0.826	0.915	1.089×
LavaMD (SoAoS 2)	4.041	3.967	0.589×
LavaMD (SoA)	2.810	3.197	0.706×
LavaMD (SoAoS 1)	0.643	0.791	1.014×

TABLE IV
PERFORMANCE RESULTS ON SELECTED BENCHMARKS AND DATA LAYOUTS ON FERMI, RELATIVE TO AOS.

Benchmark + Data Layout	Estimated Cost Against AoS	Dynamic Cost Against AoS	Speedup Against AoS
K-means (SoAoS)	0.667	0.692	1.416×
K-means (SoA)	0.333	0.385	1.570×
N-body (SoAoS 2)	1.004	1.010	0.973×
N-body (SoA)	0.950	0.950	1.018×
N-body (SoAoS 1)	0.950	0.950	1.022×
LavaMD (SoAoS 2)	5.239	5.871	0.443×
LavaMD (SoA)	2.241	2.256	0.714×
LavaMD (SoAoS 1)	0.625	0.623	1.147×

TABLE V
PERFORMANCE RESULTS ON SELECTED BENCHMARKS AND DATA LAYOUTS ON KEPLER, RELATIVE TO AOS.

From both tables, we can see our cost function and estimated cost vector can accurately predict the benefit among different data layouts on different GPU architectures. Most of the errors between estimated cost and dynamic cost come from our assumption about warp scheduling. Moreover, ignoring the GPU's ability to overlap computation with communication will also simplify the actual warp scheduling.

For example, for the LavaMD benchmark, the difference between the estimated cost and the dynamic cost is much larger than the other two benchmarks. We believe the reason is because there are more arithmetic operations in LavaMD, so the warp scheduling might be more complicated than our assumption. Furthermore, from the results, our cost estimation works better on Kepler than Fermi. We

believe the reason is that Kepler has a faster processing unit than Fermi, so the ratio of cycles of arithmetic operations over memory access operations on Kepler will be smaller than Fermi. Thus, our assumption about ignoring the arithmetic operations will have less impact on Kepler.

VI. Related Work

A. Cost Analysis on GPU

The general cost analysis on GPU is well studied. In particular, Govindaraju et al. [24] discussed the cost of GPU cache in details. Hong et al. [25] proposed an analytical model to estimate the execution time of GPU kernels, and they discussed the concept of memory coalescing in details. Zhang et al. [26] proposed a cost model that considers the efficiency of global memory access together with the shared memory access and instruction pipeline.

However, some of the above algorithms are making assumptions on the input program, and might tolerate small errors in the estimation. In contrast, we developed our cost function only based on the GPU memory hierarchy, and made little assumption on the input program. Moreover, our cost estimation enables compile time estimation, which is especially good for the data layout selection problem.

B. Data Layout Selection on GPUs

Most of the existing automatic GPU data layout selection projects are using distance-based approach. Here the term 'distance' is referring to the instruction distance defined in Section IV-B. The main idea of these approaches is defining the instruction distance between two instructions as the amount of unique memory spaces between these two instructions, and the cost between two fields as the maximum instruction distance among all closest pair of instructions accessing these pair of fields. Then the field pairs whose cost below a threshold are candidates to be merged into the same struct.

Existing distance based approaches make the same assumptions our cost model did:

1) LRU replacement policy for L1 and L2.
2) Threads are executing as in a big warp. There is no difference among the formula to calculate the distance across different threads.
3) Ignoring the ability of GPU to overlap the computation with communication, as well as the possible utilization of shared memory.

However, existing distance based approaches make additional assumptions relative to our approach. Without these assumptions, our cost analysis would be able to handle more general cases accurately.

1) Most related works use one 'distance' for the L1 and the L2 cache, or just ignoring the benefit of the L2 cache. We define L1 and L2 instruction distances separately, as the threads sharing the L1 and the L2 cache are organized differently. Moreover, the benefit of hitting in the L1 and the L2 cache are also different.
2) Most related works do not look into the array indices of each access, or assume the array indices always fall into the same cache line, i.e. they ignore the index accordance defined in Section IV-B. This assumption is not true in general, and it is sometimes unreasonable.

In particular, Mei et al. [27] discusses the benefit of different data layouts such as AoS, SoA, SoAoS, and hybrid data layout. Majeti et al. [8] uses a greedy approach to select the best SoAoS based on the instruction distance, while their work also inspected the data layout transformation between multiple kernels. Kofler et al. [9] uses a minimum spanning tree construction method to do the field partitioning based on the instruction distance, while they also enabled array tiling based on the profiling runs of the kernel with different tile sizes. Weber et al. [28] proposed an adaptive data layout selector based on the prediction function which predicts the cache behavior of GPU. Moreover, Dymaxion which developed by Che et al. [29], provides a series of APIs which enables automatic data layout transformation on GPU.

VII. Conclusion and Future Work

In this work, we proposed a novel cost analysis scheme for the data layout selection on GPUs. We built our cost function based on the cache hierarchy and memory access semantics of GPU, in order to maximize the accuracy of the cost model. We also developed an algorithm to estimate the cost of a given data layout at compile time, which mainly simulates the LRU behavior of the GPU cache. We also introduced our novel cost vector representation to handle variable loop lengths at compile time. We tested our cost estimation using selected benchmarks on Nvidia Fermi and Kepler GPU. Our performance result shows our cost model can accurately predict the efficiency of a given data layout.

Currently, we are developing an automatic data layout selector for CUDA kernels. The data layout selector extends the existing distance based field merging algorithm, and it follows the cost model proposed in this paper. We are also proposing extending the cost model to other heterogeneous architectures such as Intel Xeon Phi co-processor.

References

[1] Jack W Davidson and Sanjay Jinturkar. Memory access coalescing: a technique for eliminating redundant memory accesses. In *ACM SIGPLAN Notices*, volume 29, pages 186–195. ACM, 1994.

[2] C Nvidia. Nvidia's next generation cuda compute architecture: Fermi. *Comput. Syst*, 26:63–72, 2009.

[3] C Nvidia. Nvidias next generation cuda compute architecture: Kepler gk110. Technical report, Technical report, Technical report, 2012.[28]< http://www. top500. org, 2012.

[4] Maxwell tuning guide :: Cuda toolkit documentation. http://docs.nvidia.com/cuda/maxwell-tuning-guide/.

[5] HW Becker and John Riordan. The arithmetic of bell and stirling numbers. *American journal of Mathematics*, 70(2):385–394, 1948.

[6] Chen Ding and Ken Kennedy. Inter-array data regrouping. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 149–163. Springer, 1999.

[7] Ken Kennedy and Ulrich Kremer. Automatic data layout for high performance fortran. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 76. ACM, 1995.

[8] Deepak Majeti, Kuldeep S Meel, Rajkishore Barik, and Vivek Sarkar. Automatic data layout generation and kernel mapping for cpu+ gpu architectures. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 240–250. ACM, 2016.

[9] Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic data layout optimizations for gpus. In *European Conference on Parallel Processing*, pages 263–274. Springer, 2015.

[10] I-Jui Sung, Geng Daniel Liu, and Wen-Mei W Hwu. Dl: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*, pages 1–11. IEEE, 2012.

[11] Justin Luitjens and Steven Rennich. Cuda warps and occupancy. *GPU Computing Webinar*, 11, 2011.

[12] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[13] Paulius Micikevicius. Cuda optimization. Supercomputing, 2009.

[14] What is the difference between dram_read_transactions and gld_transactions in cuda profiler? http://stackoverflow.com/questions/27366359/what-is-the-difference-between-dram-read-transactions-and-gld-transactions-in-cu.

[15] Inside fermi: Nvidia's hpc push. http://www.realworldtech.com/fermi/8/.

[16] Fermi l2 cache hit latency? http://stackoverflow.com/questions/6744101/fermi-l2-cache-hit-latency.

[17] Fermi l2 cache how fast is the l2 cache? how do i access it? https://devtalk.nvidia.com/default/topic/496975/cuda-programming-and-performance/fermi-l2-cache-how-fast-is-the-l2-cache-how-do-i-access-it-/.

[18] Nvidia cuda library: cudagetdeviceproperties. https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/group__CUDART__DEVICE_g5aa4f47938af8276f08074d09b7d520c.html.

[19] How to determine number of register per thread. https://devtalk.nvidia.com/default/topic/494306/how-to-determine-number-of-register-per-thread-how-to-determine-number-of-register-per-thread-from-a/.

[20] Nvidia. Tesla m2050 / m2070 gpu computing module. http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_M2050_M2070_Apr10_LowRes.pdf.

[21] Nvidia. Tesla k20 gpu accelerator. http://www.nvidia.com/content/pdf/kepler/tesla-k20-passive-bd-06455-001-v07.pdf.

[22] Lars Nyland, Mark Harris, and Jan Prins. Fast n-body simulation with cuda. *GPU gems*, 3(1):677–696, 2007.

[23] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[24] Naga K Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A memory model for scientific algorithms on graphics processors. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 6–6. IEEE, 2006.

[25] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[26] Yao Zhang and John D Owens. A quantitative performance analysis model for gpu architectures. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 382–393. IEEE, 2011.

[27] Gang Mei and Hong Tian. Performance impact of data layout on the gpu-accelerated idw interpolation. *arXiv preprint arXiv:1402.4986*, 2014.

[28] Nicolas Weber, Sandra C Amend, and Michael Goesele. Guided profiling for auto-tuning array layouts on gpus. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, page 9. ACM, 2015.

[29] Shuai Che, Jeremy W Sheaffer, and Kevin Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 13. ACM, 2011.