Automatic Data Layout Generation and Kernel Mapping for CPU+GPU Architectures

Deepak Majeti
Rice University, USA
deepak@rice.edu

Kuldeep S. Meel
Rice University, USA
kuldeep@rice.edu

Rajkishore Barik
Intel Corporation, USA
rajkishore.barik@intel.com

Vivek Sarkar
Rice University, USA
vsarkar@rice.edu

Abstract

The ubiquity of hybrid CPU+GPU architectures has led to renewed interest in automatic data layout generation owing to the fact that data layouts have a large impact on performance, and that different data layouts yield the best performance on CPUs vs. GPUs. Unfortunately, current programming models still fail to provide an effective solution to the problem of automatic data layout generation for CPU+GPU processors. Specifically, the interaction among whole-program data layout optimizations, data movement optimizations, and mapping of kernels across heterogeneous cores pose a major challenge to current programming systems. In this paper, we introduce a novel two-level hierarchical formulation of the data layout and kernel mapping problem for modern heterogeneous architectures. The bottom level formulation deals with the data layout problem for a parallel code region on a given processor, which is NP-Hard, and we provide a greedy algorithm that uses an affinity graph to obtain approximate solutions. The top level formulation targets data layouts and kernel mapping for the entire program for which we provide a polynomial-time solution using a graph-based shortest path algorithm that uses the data layouts for the code regions (sections) for a given processor computed in the bottom level formulation. We implement this data layout transformation in the new Heterogeneous Habanero-C (H2C) parallel programming framework and propose performance models to characterize the data layout impact on both the CPU and GPU. Our data layout framework shows significant performance improvements of up to 2.9× (geometric mean 1.5×) on a multicore CPU+GPU compared to the manually specified layouts for a set of parallel programs running on a heterogeneous platform consisting of an Intel Xeon CPU and an NVIDIA GPU. Further, our framework also shows performance improvements of up to 2.7× (geometric mean 1.6×) on just the multicore CPU, demonstrating the applicability of our approach to both heterogeneous and homogeneous hardware platforms.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques

Keywords Heterogeneous architectures, data layout, kernel mapping

CC'16, March 17–18, 2016, Barcelona, Spain
ACM. 978-1-4503-4241-4/16/03...
http://dx.doi.org/10.1145/2892208.2892210

1. Introduction

The end of Dennard scaling has brought about a drastic change in the processor design landscape over the last decade. Since we can no longer expect higher clock speeds within a reasonable power budget, we are entering an era of heterogeneous and specialized processors: a trend expected to continue in the future. Heterogeneous architectures have processors that exhibit wide diversity in features ranging from the number of processor cores to the memory hierarchy structure. For example, one of the dominant heterogeneous architectures in use today is a CPU+GPU system with the CPU containing a small number of "fat" cores, and the GPU containing a much larger number of "thin" cores. Furthermore, the memory hierarchy and cache structures are very different on the CPU and the GPU. Such diverse characteristics of heterogeneous architectures not only make the portability a difficult task but also make performance optimization very challenging.

A key task in the optimization process is determining the layout of data items in memory for a given application and architecture. Recent studies [11, 30, 34, 38] have shown that data layouts play a major role in determining application performance on both the CPU and GPU, and that different data layouts yield the best performance on CPUs vs. GPUs. For example, CPUs usually perform well with an Array-Of-Struct (AoS) layout because an AoS layout can help improve prefetching and cache sharing on CPUs while avoiding false sharing. On the other hand, GPUs usually perform well with a Struct-Of-Array (SoA) layout since a SoA layout can improve the performance on GPUs due to coalescing of memory accesses. Determining the optimal data layout, however, remains a challenging task in general, since the performance of a data layout depends on factors such as (a) number of parallel hardware threads/contexts available; (b) memory hierarchy; (c) data access pattern in the program; (d) input size of the program.

With the ubiquity of heterogeneous CPU+GPU systems in various domains such as mobile and server, many heterogeneous parallel programming models have emerged recently, e.g., OpenCL [19], CUDA [5], OpenACC [20], C++AMP [1], Lime [7], and Concord [8]. While the above models may differ on their abstraction level, i.e. low-level such as OpenCL, CUDA or high-level such as OpenACC, C++AMP, Concord; all of them require that data layouts be specified by the programmer; which is a major stumbling block for programmer productivity, portability, and performance tuning. It is, therefore, important that optimizing compilers targeting heterogeneous CPU+GPU systems should perform data layout transformations automatically. The inability of current state-of-the-art techniques to handle data layout generation was recently summarized by Norm Rubin from NVIDIA in his keynote talk at PPoPP-2014: "*As parallelism goes up, the memory interconnect gets more complex so layout matters, but it is up to the programmer*". In this paper, we propose an approach to address this problem by introducing new automatic data layout generation techniques for heterogeneous architectures.

Another challenge with these heterogeneous systems is to find the best mapping for a given program onto the underlying hardware. The mapping problem involves (1) identifying the performance of code blocks (kernels) on different processors (e.g. CPU, GPU) and then (2) identifying the optimal mapping for the entire program while considering data movement costs among the processors. Since data layout impacts the performance on a given processor, the data layout and mapping problems are inter-dependent.

The data layout problem was studied earlier in the context of multi-core CPU execution [12, 17, 18, 22, 23, 30, 31]. These approaches cannot be easily extended to programs executing on a heterogeneous CPU+GPU system since the past work focuses on identifying a single best layout for the entire program on a single processor type. The mapping problem has been recently studied in the context of CPU+GPU architectures [14, 24, 28]. However, these approaches do not consider the data layout of the program.

The best layout for a given program could either be a single layout for the entire program or different layouts for different parts of the program, with data remapping in between them. Additionally, some data-parallel kernels in a program may be well-suited for CPU execution while others may execute faster on GPUs. In these scenarios, the cost of data communication between the CPU and GPU also plays a major role in deciding the best data layout to achieve optimal performance. In our work, we address these issues and introduce a novel scheme that automatically generates the best data layout and kernel mapping for a program on a heterogeneous CPU+GPU architecture.

Specifically, our contributions are as follows:

- We introduce a novel two-level hierarchical formulation for the problem of data layout and kernel mapping for a given program. This formulation, unlike previous work, allows us to separate complexity theoretical harder steps from easier steps in data layout and mapping generation.
- We develop an automatic data layout compiler transformation that implements the two-level hierarchy formulation described in the Heterogeneous Habanero-C (H2C) compiler that targets CPU+GPU processors. First, we build affinity graphs from the Parallel Intermediate Representation (PIR) of the input program. We then use these affinity graphs along with the cost of remapping from one layout to another to automatically determine the best data layout for a given program.
- We present an experimental evaluation of our automatic data layout transformation on a CPU+GPU platform (Intel Xeon CPU + NVIDIA GPU) using a diverse set of 7 benchmarks. Our results show that the automatic framework improves the performance up to 2.7× (geometric mean 1.6×) on a homogeneous multicore CPU and up to 2.9× (geometric mean 1.5×) on a heterogeneous CPU+GPU compared to the manually specified layouts.

The rest of the paper is organized as follows. In Sec. 2, we explain the data layout and mapping problem using an example program. Sec. 3 introduces a mathematical formulation of the problem. We discuss the implementation details of our framework in Sec. 4. The experimental evaluation is described in Sec. 5. In Sec. 6, we discuss related work. Finally, we conclude in Sec. 7.

2. Motivating Example

In this section, we demonstrate through a motivating example how the choice of data layout and kernel mapping can lead to significant performance gains on a heterogeneous CPU+GPU architecture. Furthermore, we illustrate complexity and intricacies in selecting the best data layout for a given architecture. Let us consider a micro-benchmark with two data-parallel loops as illustrated in Figure 1. We use H2C's forasync syntax for the data-parallel loops (details of H2C are given in Section 4.1). The clauses in the forasync loops are as follows: *point* specifies the loop's index variable, *range* describes the iteration domain (M = 10240×10240 in this example) and *at* specifies the target device ("NVIDIA Kepler K40C" or "Intel Xeon CPU" in our case). The first data-parallel kernel implements a stencil-like computation involving 5 arrays, x, y, z, w, e, and the second kernel executes a simple multiply and add computation involving 4 arrays x, y, z, e.

We execute the program on the NVIDIA GPU and Intel CPU with two different layouts: AoS with x, y, z, w in a structure and SoA where each of these fields are individual structures (arrays). Kernel-1 take 10 msec with AoS, 22 msec with SoA on the GPU and 62 msec with AoS, 105 msec with SoA on the CPU. Kernel-2 takes 15 msec with AoS, 6 msec with SoA on the GPU and 12 msec with AoS, 7 msec with SoA on the CPU. Remapping from the AoS layout to SoA layout takes 4 msec on the GPU and moving data between CPU and GPU takes around 3 msec per field. If the data layout and kernel mapping choices are left to the programmer, then the best performance that can be achieved is 45 msec. The order would be (1) copy from CPU to GPU(14 msec), (2) kernel-1 with AoS-GPU(10 msec), (3) kernel-2 AoS-GPU(15 msec), (4) copy from GPU to CPU (6 msec). However, the optimal order is (1) copy from CPU to GPU(14 msec), (2) execute the first kernel on the GPU with AoS layout (10 msec), (3) remap the data from the AoS to the SoA(4 msec),(4) copy data from GPU to the CPU (3 msec), and (5) finally execute the second kernel on the CPU with the SoA layout(7 msec). The application now takes the best execution time of 38 msec resulting in a speedup of 1.18. Therefore, a single data layout and mapping all the kernels to a single processor is not optimal in this case.

```
struct ABCD{float x; float y; float z; float w;};
float ∗x, ∗y, ∗z, ∗w, ∗e;
initialize(x, y, z, w, e); // on CPU
// Copy x, y, z, w, e from CPU to GPU: 14 msec
// Kernel−1  AOS(msec)  SOA(msec)
//  GPU         10          22
//  CPU         62          105
finish forasync point(i) range(0:M) at(dev){
 if(.....){
   e[j] = ((x[j] + y[j] + z[j]) / w[j])
       + ((x[j+1] + y[j+1] + z[j+1]) / w[j+1])
       + ((x[j+2] + y[j+2] + z[j+2]) / w[j+2])
       + ((x[j+3] + y[j+3] + z[j+3]) / w[j+3]);
 }
}
// Remap from AoS to SoA: 4 msec
// Copy e from GPU to CPU: 3 msec
remap(xyzw, x, y, z, w);
// Kernel−2  AOS(msec)  SOA(msec)
//  GPU         15          6
//  CPU         12          7
finish forasync point(i) range(0:M) at(dev){
   x[j] = (y[j] + e[j] ∗ 1.432);
   z[j] = (x[j] + e[j]);
}
// Copy x, z from GPU to CPU: 6 msec
```

Figure 1: Microbenchmark in *H2C*. Best execution is when Kernel-1 executes with AoS layout on the GPU, followed by data remapping from AoS to SoA and then Kernel-2 executes with SoA layout on the CPU.

It is interesting to observe that while popular practice is to use a SoA to achieve coalesced memory accesses, we instead discover that AoS layout on GPU is more beneficial in certain cases. On the GPU, the AoS layout is specified using aligned structures such as *float2* and *float4* types. When we profiled the above code using an NVIDIA profiler [27], we observed that the compiler generated **128-bit** loads for float4 types, **64-bit** loads for float2 types and **32-bit** loads for *float* types. The benefit from 128-bit loads comes from the fact that there are fewer instructions to issue (compared to 4 32-bit loads). Therefore, we noticed that as long as the fields are always accessed together, it is better to arrange them in an AoS layout, which was also observed in [26].

In summary, the above example shows how the choice of data layout can lead to significant performance gains. Furthermore, the approaches based on conventional wisdom might lead to sub-optimal data layout and kernel mapping choices. In this context, we propose an automatic data layout generation and kernel mapping for CPU+GPU architectures.

3. Problem Formulation

In this section, we formalize the optimal data layout and kernel mapping problem and provide corresponding complexity results. The objective of our framework is to automatically determine the best data layout(s) and kernel mapping for a given CPU+GPU architecture and generate the corresponding executable. As illustrated in the previous section, due to the variations in data access patterns across code regions in a program, a single layout for the entire program may not be always optimal. In the following subsections, we propose a scheme that produces different data layouts for different parts of the program and automatically infers a mapping.

3.1 Hierarchical Approach

To assign different data layouts at different points in a program, we need a mechanism to partition the program. To this end, we treat data parallel kernels as the smallest unit of the program and partition the program into disjoint sections[1] and initially assign a single data-parallel kernel for each section. For simplicity of exposition, in our theoretical analysis, we assume all sections lie in a single control flow path (i.e. there are no branches). The subsequent implementation, however, deals with control flow between sections as discussed in Sec 4.2.

We propose a novel two-level hierarchical formulation of the data layout and mapping problem. The bottom level formulation, Section Data Layout (SDL), deals with the data layout selection for a section based on interactions within a section. SDL is applied for each processor type. On the other hand, the top level formulation, Program Data Layout (PDL), takes in data layouts computed at the SDL level and computes the data layout and kernel mapping for the overall program.

We first discuss PDL and prove that PDL can be computed in polynomial time. Then we move on to the bottom level SDL, which is NP-hard. To address the intractability of SDL in practice, we propose a greedy algorithm that is later employed in our experiments.

Let S $= \{S_1, S_2, \cdots S_n\}$ be the set of sections for a program P. We denote the set of fields of P by F such that $F = \{f_1, ..., f_r\}$. To avoid notational clutter, we use *field* to refer to the fields in both AoS and SoA (which are actually arrays). Accordingly, the data layout D $= \{d_1, d_2, \cdots d_n\}$ and E $= \{e_1, e_2, \cdots e_n\}$ represent the corresponding data layouts of fields and kernel mapping for each section respectively. We assume that the set of fields in data layout for section S_i is a subset of F^i such that $F^i = f_1 \cup f_2 \cdots f_i$. It is worth noting that our complexity results do not change if we assume the complement of the above assumption i.e. $F^i = f_i \cup f_{i+1} \dots f_n$. Let $\mathsf{Cf}(d_i, S_i, e_i)$ denote the cost of executing section S_i with data layout d_i on processor e_i, $C(d_i, d_{i+1})$ to denote the cost to obtain data layout d_{i+1} from d_i.

[1] We apologize to the reader for overloading the word "section". We henceforth use "Sec." refer to a Section in the paper's organization structure

3.2 Program Data Layout

The problem of Program Data Layout (PDL) is concerned with the selecting of data layout and mapping for the entire program while considering inter-section interactions. PDL takes in the data layouts returned by SDL for each section per processor type and returns the data layout and mapping for the entire program. For the sake of simplicity, we will first describe the PDL framework for a single processor type. This will handle the data layout for the entire program on a single processor type. We will then extend the framework to handle the mapping of the entire program for multiple processor types.

The control flow among sections allows us to construct a directed acyclic graph with in-degree and out-degree of nodes restricted to at most 1. As discussed above, the data layout for a section can consist of fields accessed by its predecessors. To facilitate this, we introduce an operation combine that takes in optimal data layouts d_i, d_j for sections S_i, S_j such that S_j is successor of S_i and returns the data layout by merging d_i, d_j. In other words, the combine operation results in a data layout that is best for both S_i and S_j sections collectively. We use $cost(\mathsf{combine}(d_i, d_j))$ to represent the cost of the combine operation for data layouts d_i and d_j.

Another possible operation is remap which remaps the data layout from d_i to d_j. The cost for remap is directly proportional to the number of fields between data layouts that are remapped. We use

$$\mathsf{cost}(d_1^f, d_2^i, d_2^f) = \begin{cases} cost(\mathsf{combine}(d_1^f, d_2^i)) & \text{if } d_2^f = \mathsf{combine}(d_1^f, d_2^i) \\ cost(\mathsf{remap}(d_1^f, d_2^f)) & \text{otherwise} \end{cases}$$

to denote the cost of transformation of d_2^i to d_2^f where d_1^f is the data layout of the preceding section. Therefore, using the notation introduced in Sec. 3.1 we have $C(d_1^f, d_2^f) = \mathsf{cost}(d_1^f, d_2^i, d_2^f)$.

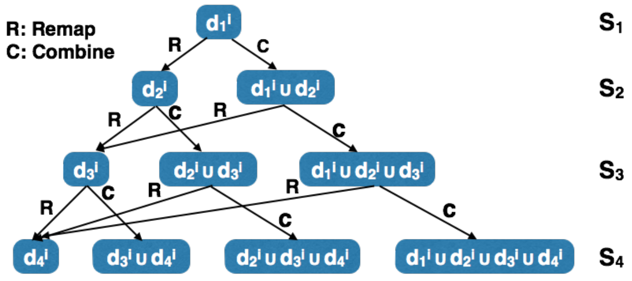


Figure 2: PDL configurations with combine and remap edges for a single processor type

For example: considering a single processor, for $n = 4$, we have sections S_1, S_2, S_3, S_4 and data layouts returned by SDL is $\{d_1^i, d_2^i, d_3^i, d_4^i\}$, where subscript i is used to denote the input to to PDL (We use superscript f to denote the "final" data layout returned by PDL). One possible final configuration is

$$d_1^f = d_1^i; d_2^f = \mathsf{combine}(d_1^i, d_2^i); d_3^f = d_3^i; d_4^f = \mathsf{combine}(d_3^i, d_4^i);$$

and the cost associated with it is

$$\mathsf{cost}(\mathsf{combine}(d_1^i, d_2^i)) + \mathsf{cost}(\mathsf{remap}(\mathsf{combine}(d_1^i, d_2^i), d_3^i))$$
$$+ \mathsf{cost}(\mathsf{remap}(d_3^i, d_4^i)) + \sum_{i=1}^{4} \mathsf{Cf}(d_1^f, S_i, e_i)$$

Figure 2 illustrates all the possible configuration for this case. The four different final layouts possible for section S_4 are shown below

$$d_4^f = d_4^i$$
$$d_4^f = \mathsf{combine}(d_3^i, d_4^i)$$
$$d_4^f = \mathsf{combine}(\mathsf{combine}(d_2^i, d_3^i), d_4^i)$$
$$d_4^f = \mathsf{combine}(\mathsf{combine}(\mathsf{combine}(d_1^i, d_2^i), d_3^i), d_4^i)$$

Every possible data layout can be specified by the last remap operation. For example, in case of d_4^i, the last remap was applied at the section 3 and for combine(d_3^i, d_4^i), the last remap operation was applied at the section 2.

When multiple processors are included, there could be an additional copy cost involved to copy the data between these processors. Let copy(e_i, e_{i+1}) be the cost to copy data between processor e_i and e_{i+1}. Figure 3 shows the PDL configurations when multiple processor types are present.

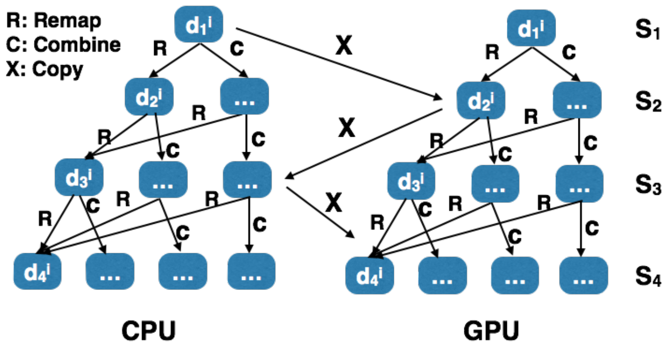


Figure 3: PDL configurations on a CPU+ GPU architecture with copy edges

We formulate the Program Data Layout (PDL) problem as follows: a PDL takes in the set of data layouts $\{d_1^i, d_2^i, \cdots d_n^i\}$ computed per processor type from SDL and returns a set of data layouts $\{d_1^f, d_2^f, \cdots d_n^f\}$ such that the computed cost

$$\sum_{i=1}^{n-1} C(d_i^f, d_{i+1}^f) + \sum_{i=1}^{n} \mathsf{Cf}(d_i^f, S_i, e_i) + \sum_{i=1}^{n-1} \mathsf{copy}(e_i, e_{i+1})$$

is minimum.

The following theorem presents the complexity analysis of PDL.

THEOREM 3.1. PDL *is in PTIME.*

Proof To prove PDL is in PTIME, we reduce PDL to finding the shortest path over a graph. To this end, we construct a DAG for every $G = (V, E)$ where a node represents a possible data layout for a Section. We call this DAG the PDL-DAG. From above we know that for section S_i, there are only i possible data layouts. In our DAG, an edge represents either combine or remap operation. Let $D_{i,j}(i > j)$ represent the final data layout for section i obtained such that the last remap operation was at section j. Also, we obtain $D_{i+1,j}$ and $D_{i+1,i}$ by applying combine and remap operations respectively. Therefore in our DAG G, $V = \{D_{i,j|0\le j<i<n}\} \cup D_{\text{dest}}$, where n is the total number of sections and D_{dest} is an extra node we introduce for technical reasons explained later. We construct all the combine and remap edges such that the weight of combine edge $(D_{i,j}, D_{i+1,j})$ is sum of the cost of combine edge and $\mathsf{Cf}(D_{i+1,j}, S_{i+1}, e_i)$. The edges from $D_{n,j|0<j<n}$ to D_{dest} are added with weight 0. Therefore, $E = \{(D_{i,j}, D_{i+1,j})\} \cup \{(D_{i,j}, D_{i+1,i})\} \cup \{(D_{n,j}, D_{\text{dest}})\}$ for $0 \le j < i < n$. With this formulation, the problem PDL reduces to finding the shortest (weighted) path from $D_{1,0}$ to D_{dest}. The shortest path for this graph can be computed in $\mathcal{O}(|E| + |V| \log |V|)$.

We now compute the cardinalities of sets V and E. For section S_i we have i nodes in G. Therefore summing up all the nodes and adding 1 for D_{dest} node we have $|V| = 1 + \sum_{i=1}^{n} i = 1 + n(n+1)/2$. Also, for every node $D_{i,j}(i < n)$, we have 2 outgoing edges and for nodes $D_{n,j}$ we have one outgoing edges. Thus summing up all the edges, we have $|E| = n + \sum_{i=1}^{n-1}(2 * i) = \mathcal{O}(n^2)$. Note that the number of processor types are constant. Therefore, the shortest path for G can be computed in $\mathcal{O}(n^2 + n^2 \log n) \in \mathcal{O}(n^2 \log n)$. Hence, the problem PDL can be computed in PTIME.

3.3 Section Data Layout

The objective of SDL is to find the data layout for a given section, considering only Array of Structure (AoS) and Structure of Array (SoA) layouts. In any instance of a data layout, there is a single SoA but multiple AoS possible. Figure 4 shows an instance of the data layout possible for a section which uses 7 fields $\{*a, *b, *c, *d, *e, *f, *g\}$.

```
struct SOA{float *a; float *b;} ab;
struct AOS1{float c; float d;} cd[100];
struct AOS2{float e; float f; float g;} efg[100];
```

Figure 4: Example data layout instance

Based on the code and the target architecture, affinity values are associated with every pair of fields. The computation of affinity values is discussed in detail in Sec 4. The fields and the affinity values can be represented as a weighted complete graph $G_{\text{cluster}} = (V, E)$, where $V = F$ and $(v_1, v_2) \in E$ for every $v_1, v_2 \in V$. Let $\mathsf{W}(e) \in N$ denote the weight of edge e and $\mathsf{W}(G = (V, E))$ denote sum of weights for all the edges $e \in E$. An optimal data layout would combine fields into cluster such that the sum of weights of inter-cluster edges would be minimum, therefore sum of weights of clusters edges to be maximum. This stems from the observation that sum of weights of inter-cluster edges is proportional to cache misses. Due to factors such as pre-fetch size, the size of every cluster is bounded to a given constant, henceforth denoted as k. Therefore, optimal data layout problem for a section, denoted as SDL, can be formulated as follows:

SDL(G, k): Given a weighted complete graph $G_{\text{cluster}} = (V, E)$ with integer weights, find a partition $OC = \{C_1, C_2, ...C_i\}$ such that $|C_i| < k$ and $\sum \mathsf{W}(C_i)$ is maximum. The SDL problem is NP-hard [13, 18].

3.4 Greedy Strategy

While the NP-hardness of SDL motivates us to ask if approximation to SDL is easier, the complexity analysis of approximation to SDL is beyond the scope of this paper and requires a further study. We instead propose an algorithm, SGML, based on greedy-heuristic strategies. On a high-level, the algorithm sorts the edges according to their weights and has flavor of the union-find algorithm. The pseudo-code for the algorithm is presented in Algorithm 1. SGML takes in two parameters as input: an affinity graph $G = (V, E)$ and an integer k, which bounds the maximum size of a cluster. SGML assumes access to three subroutines: (1) CreateNewCluster takes as input a pair of two nodes (u, v) and returns a new cluster that contains u and v, (2) AddToCluster takes as inputs a cluster c_u and a node v and adds node v to the cluster c_u, (3) MergeClusters takes as inputs two clusters c_u and c_v, and merges cluster c_v into c_u. SGML chooses the edges in decreasing order of their weights. For every edge (u, v) chosen, there are five possibilities: (1) u and v do not belong to any of the clusters: in this case, a new cluster

Algorithm 1 SGML (G = (V, E), k)

```
1:  $E' \leftarrow \mathsf{WeightSorted}(E)$
2:  $C = \{\}$
3:  for edge $e = (u, v)$ in $E'$ do
4:      $c_u = FindCluster(u); c_v = FindCluster(v)$
5:      if ($c_u ==$ NULL && $c_v ==$ NULL ) then
6:          $c = \mathsf{CreateNewCluster}(u, v); C = C \cup c$
7:      else if ($c_v == NULL$ && $|c_u| < k - 1$) then
8:          $\mathsf{AddToCluster}(c_u, v))$
9:      else if ($c_u == NULL$ && $|c_v| < k - 1$) then
10:         $\mathsf{AddToCluster}(c_v, u)$
11:     else if ($c_u! = c_v$ && $|c_u| + |c_v| < k$) then
12:         $\mathsf{MergeClusters}(c_u, c_v)$
13: return C
```

with the vertices u and v is created, (2) u does not belong to any cluster and the size of cluster for $v(c_v)$ is less than $k - 1$: in this case, we add u to c_v, (3) v does not belong to any cluster and the size of the cluster for $u(c_u)$ is less than $k - 1$: in this case, we add u to c_v, (4) u and v belong to different clusters (c_u and c_v respectively) such that $|c_u| + |c_v| < k$, in this case we merge clusters c_u and c_v and (5) for cases not covered above, we ignore the edge and proceed to the next edge. We run the SGML algorithm for both CPU and the GPU. A complexity-theoretical analysis of SGML is beyond the scope of this paper and left for future work.

4. Automatic Data Layout Framework in H2C

In this section, we discuss the implementation details of our automatic data layout framework in the heterogeneous Habanero-C (H2C) programming system. We first briefly describe the H2C programming model that is an extension of the Habanero-C programming model [2]. We then describe our overall data layout and mapping framework that consists of a set of analysis passes followed by the data layout transformation pass. Finally, we describe the details of how affinity graphs are constructed including how remap and combine costs are computed for a H2C program.

4.1 H2C Programming Model

H2C is a high-level programming language that targets both multicore CPU and GPU architectures. The high-level parallel constructs in H2C are:

- **async** *copyin*$\langle args \rangle$ *copyout*$\langle args \rangle$ *at*$\langle device \rangle$: Asynchronously copy data specified by the arguments to and from the *device*. No code body is required.
- **forasync** *point*$\langle args \rangle$ *range*$\langle args \rangle$ *at*$\langle device \rangle$\{Body\}: Multi-dimensional data parallel loop. The loop indices are specified by the *point* clause. The loop bounds are specified by the *range* clause and the optional *at* clause is used to specify the mapping of the kernel to the devices. There is no implicit barrier at the end of the forasync construct. The programmer is responsible for ensuring that the loop iterations are logically independent and can be executed in parallel. (no ordering is assumed evenin the presence of floating-point computations).
- **finish** $\langle Body \rangle$: Ensures that *async* and *forasync* tasks spawned inside *Body* are completed.

Since data layout impacts only data-parallel kernels that target both CPUs and GPUs, we only consider forasync and finish constructs in this work. The H2C compilation framework consists of a static compiler based on the ROSE [29] infrastructure and an OpenCL-based runtime. The static compiler automatically generates host-side binary with embedded OpenCL code (for the bodies forasyncs) from high-level C code. The OpenCL code is then JIT-ed during runtime using the vendor specific OpenCL SDK and subsequently executed on the GPU. We implement our framework in the static compiler.

4.2 Data Layout Framework

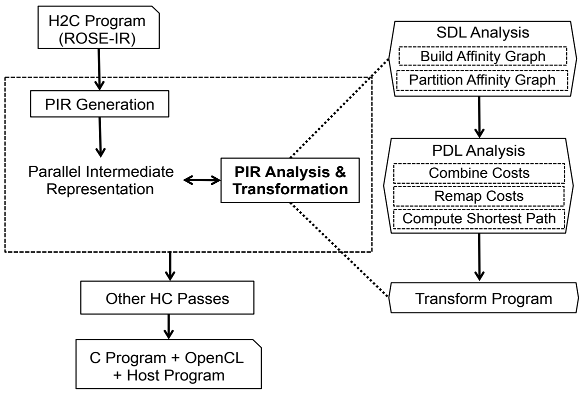


Figure 5: Compiler framework for automatic data layout

Figure 5 shows a diagrammatic description of our data layout transformation framework. From ROSE IR, we generate the parallel intermediate representation (*PIR*) [39]. Once the *PIR* is constructed, we perform data layout analysis for each data-parallel section (SDL). During SDL analysis, we build an *affinity graph* for each section and then employ the algorithm SGML described in Sec. 3.3 to partition the affinity graph. We find the best data layout for both the CPU and GPU processors. Subsequently, we perform data layout analysis for the entire program. During this phase, we compute the remap and combine costs for kernels. We find the kernel performance on each processor type by profiling. One can also include other kernel performance estimation techniques described in [24, 28]. We then apply the shortest path algorithm described in Section 3.2 to obtain the best data layout and kernel mapping for each section. The remap costs between two different processors will also include the data copy cost. The H2C compiler uses def-use analysis to determine modified data that requires copying. We use the superblock technique [36] to handle the case where there is control-flow between parallel sections of a program [2]. Finally, the program is transformed to use the above-determined layout and mapping. The placement of the remap operations are done carefully using code motion techniques described in [21]. We now discuss the construction of *PIR*, affinity graph, and computation of remap/-combine costs in more detail.

4.3 PIR

The *PIR* is a common intermediate language for explicitly parallel programs such as *H2C*. For every function in a program, the *PIR* for that method consists of three key data structures: 1) a Region Structure Tree (*RST*), 2) a set of Region Control Flow Graphs (*RCFG*), and 3) a set of Region Dictionaries (*RD*). The *RST* represents the region nesting structure of the method being compiled, analogous to the Loop Structure Tree (LST) introduced in [32]. Each region in the *RST* has an associated control flow graph (*RCFG*) that encapsulates control flow for the immediate children of the region. Additionally, each region stores summary information, such as upwards-

[2] Although we implement this feature in our compiler, we found that none of the benchmarks used in our evaluation exhibit any conditional control-flow patterns between parallel sections.

exposed uses and downwards-exposed defs, in an associated dictionary (*RD*).

For *H2C*, the single-entry regions considered in this work include `FINISH`, `FORASYNC`, and loop regions. Two special empty regions `START` and `END` are added to designate the start and end of a function. The other IR nodes considered in the *RCFG* are array load `ALOAD`, array store `ASTORE`, object field load `FLOAD`, and object field store `FSTORE`.

4.4 Affinity Graph Construction

The affinity graph construction is an important component of our framework that captures how close a group of data items are accessed together in the program. We build the affinity graph for each section. The affinity graph is a weighted undirected graph where the nodes represent individual data items (a statement of the form `ALOAD`, `ASTORE`, `FLOAD`, `FSTORE`) and edges represent the co-access pattern of two data items. The weight on an edge reflects the frequency of accessing them together and also the amount of memory accessed in between them. Following past approaches for static cost estimation, the frequency of array access inside a loop-nest is estimated as 10^d, where d denotes loop depth.

To reduce the size of the resulting affinity graph, the body of a section is heavily optimized before the construction of the affinity graph. In particular, scalar replacement is performed aggressively to eliminate accesses to $a[i-1]$ where a prior iteration loads $a[i]$ with no killing dependency in between them in a loop region. Similarly, variable renaming is performed in such a way that loops iterating over the same iteration space (exactly same lower and upper bounds) are assigned the same index variable name.

For sections consisting of accesses to both arrays and object fields, we build two separate affinity graphs: one focusing on arrays and another focusing on object fields. Note that the affinity graph for arrays must capture information about the amount of memory needed by the object fields accessed in between and vice versa. This information is conservatively computed. For the rest of the discussion, we will only focus on building the affinity graph for array accesses.

We now describe a flow-insensitive algorithm to build affinity graph as shown in Algorithm 2. We start by scanning a basic block from top to bottom. If we visit an `ALOAD` $a[i]$ or `ASTORE` $a[i]$ instruction, we create a node for $a[i]$, if it is not there already in the affinity graph. We count the number of memory accesses, mem_usage$(a[i], b[i])$, from the previous `ALOAD` $b[i]$ or `ASTORE` $b[i]$ instruction (takes into account object field accesses). We add an edge between $a[i]$ to $b[i]$ with the edge weight $w(e(a[i], b[i]))$ as:

$$w(e(a[i],b[i])) = \begin{cases} 0, & \text{if mem}(a[i],b[i]) > \text{cache\_size} \\ \text{freq}(B) * \frac{1}{log_2 \text{mem}(a[i],b[i])}, & otherwise \end{cases} \quad (1)$$

where freq(B) denotes the frequency of basic block B. If the memory usage, $mem(a[i], b[i])$, is greater than the cache size, then we assign 0 as weight indicating there is no point combining them. Otherwise, the weight is computed as the product of the basic block frequency and the inverse of the logarithm of the memory usage. It is important to emphasize the freq(B) component since frequently executed blocks will contribute significantly to the over all data layout. If the edge already exists, we accumulate the edge weights to account for aggregated frequency counts.

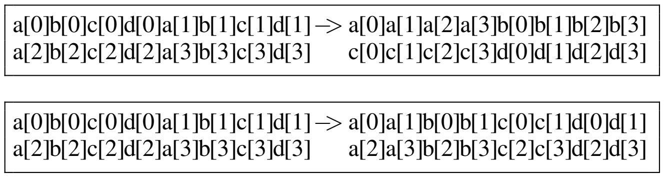
4.5 Remap Cost Estimation

The remap cost estimation not only depends on the amount of data being remapped but also depends on the type of remapping used. Different types of remapping operations are:

- *Local Data Remapping (LDR)*: remaps the data in blocks.
- *Out-of-place Global Data Remapping (OGDR)*: remaps the entire data from one data layout to another but uses an additional buffer.
- *In-place Global Data Remapping (IGDR)*: remaps the entire data from one data layout to another without any additional buffer.

Algorithm 2 Affinity graph of a parallel section

```
1:  procedure AFFINITYGRAPH(PIR: PIR for the parallel section)
2:      $V := \{\}; E := \{\}$;
3:      for each loop region L in PIR do
4:          for each basic block B in the RCFG(L) do
5:              $mem$ := 0;
6:              $prev\_I$ := {};
7:              for each instruction I in B do
8:                  if I is an FLOAD a.f or FSTORE a.f then
9:                      $mem$ += $sizeof(a.f)$;
10:                 if I is an ALOAD a[i] or ASTORE a[i] then
11:                     Create a node for $a[i]$, if not already in $V$;
12:                     if $prev\_I$ is $ALOAD\ b[i]$ or $ASTORE\ b[i]$ then
13:                         if edge between $a[i]$ and $b[i]$ is absent then
14:                             Add an edge $e$ between nodes $a[i]$ and $b[i]$
15:                         Assign/Update edge weight, $w(e)$ using Eq. 1;
16:                     $prev\_I := I$;
17:                     $mem$ := 0;
18:     return
19: end procedure
```

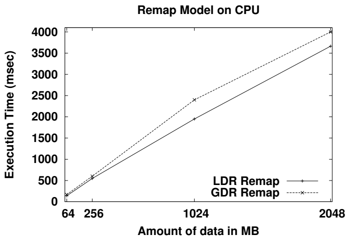

a[0]b[0]c[0]d[0]a[1]b[1]c[1]d[1] -> a[0]a[1]a[2]a[3]b[0]b[1]b[2]b[3]
a[2]b[2]c[2]d[2]a[3]b[3]c[3]d[3]   c[0]c[1]c[2]c[3]d[0]d[1]d[2]d[3]

a[0]b[0]c[0]d[0]a[1]b[1]c[1]d[1] -> a[0]a[1]b[0]b[1]c[0]c[1]d[0]d[1]
a[2]b[2]c[2]d[2]a[3]b[3]c[3]d[3]   a[2]a[3]b[2]b[3]c[2]c[3]d[2]d[3]

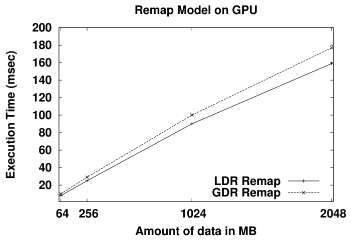
Figure 6: (Top) Global data remapping, (Bottom) Local data remapping

Although IGDR saves space, it is computationally inefficient as it requires several synchronization operations when performed in parallel. In contrast, OGDR does not require any synchronization. We focus on OGDR and LDR remappings for the rest of the discussion. OGDR transforms the entire data from AoS to SoA. LDR on the other hand regroups the data to a local SoA data layout in blocks. Figure 6 demonstrates how LDR and OGDR are constructed with the help of four arrays, $a[0:3]$, $b[0:3]$, $c[0:3]$, and $d[0:3]$. The data layout on the left-hand side is in AoS. The top-right shows the GDR version of SoA whereas the bottom-right shows the LDR version of SoA (uses a block size of 2): two elements of arrays a, b, c, d are mapped to SoA layout followed by the remaining two elements of each array.

A remap operation can be parallelized to reduce its impact on execution time. We empirically determine the remap cost for LDR and OGDR with the help of micro-benchmarks on a given hardware platform (this operation is performed once per platform and stored in a table). Figure 7 depicts the data remapping costs on a Tesla M1050 GPU (to the left) and an Intel Xeon CPU (to the right). On the X-axis, we use the amount of data being remapped. The charts show that it is always beneficial to perform remapping on the GPU as opposed to the CPU. Additionally, LDR is always faster than OGDR on both the CPU and the GPU. This is because LDR

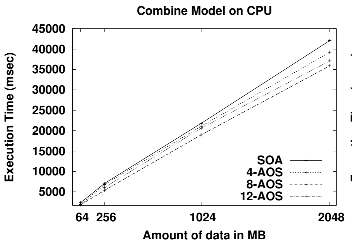


(a) CPU Model

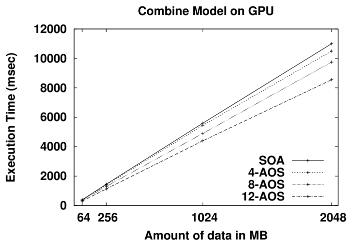


(b) GPU Model

Figure 7: Remap cost model on an NVIDIA Tesla GPU and an Intel Xeon CPU



(a) CPU Model



(b) GPU Model

Figure 8: Combine cost model on an NVIDIA Tesla GPU and an Intel Xeon CPU for a memory-bound kernel with varying AoS size

Algorithm 3 Estimate remap cost

```
 1: procedure RemapSections(S1, S2: parallel sections)
 2:     fieldsize ← 0
 3:     for f ϵ S1.Fields do
 4:         //for each field or array
 5:         if f.getLayout(S2)==NULL then
 6:             //if f is accessed in one but not in another
 7:             continue;
 8:         if f.getLayout(S1) $neq$ f.getLayout(S2) then
 9:             // the layouts are different
10:             // combine the frequency of the basic block
11:             // containing the field or array
12:             fieldsize$+ = f.size * freq(basicblock(f))$;
13:     return remap_model(fieldsize) + copy_cost(fieldsize);
14: end procedure
```

benefits from data locality on the CPU whereas, on the GPU, it performs remapping by taking advantage of scratchpad memory and *local barrier*. On the other hand, LDR is feasible only when the same partition of data items are remapped across multiple kernels. In our evaluation, by default we use LDR to remap the data except for the case where two consecutive kernels remap data from different partitions (as computed using SGML algorithm), at which point we switch to OGDR.

Algorithm 3 presents the remap cost estimation. It takes two parallel sections as input and outputs the estimated cost of remapping. The algorithm iterates over the fields (both object fields and array accesses) in both sections, checks if a field appears in only one of the section's data layout (and not in both) and accordingly updates the counter fieldsize, which counts the amount of data that needs to be remapped. This value is passed to the remap_model (as shown in Figure 7) which then returns the cost of remapping. A copy cost based on the fieldsize value also gets added to the final cost of the remap edge if the sections under consideration are mapped to different processors. Note that the SDL phase gives us best data layouts per section on each processor.

4.6 Combine Cost Estimation

The combine cost is estimated as the loss in performance by assigning the same data layout for two sections instead of the previously assigned individual data layouts. If the layouts of both the sections are the same, then the combine cost is 0. If the layouts are different, then an intermediate layout $DL12$ is obtained by combining the two sections, $S1$ and $S2$, and running the SGML algorithm on the combined affinity graph $S12$.

The combine cost is the predicted performance loss and is the sum of the difference between running the sections with the original layouts $DL1, DL2$ compared to running them using the new layout $DL12$. The pseudo-code for the procedure CombineSections is presented in Algorithm 4.

Algorithm 4 Compute combine cost

```
 1: procedure CombineSections(S1, S2: parallel sections)
 2:     // merge affinity graphs for S1 and S2
 3:     // perform partitioning using SGML algorithm
 4:     DL12 = SGML(merge(S1.affinity_graph,S2.affinity_graph))
 5:     // cost of executing S1 using the combined layout DL12
 6:     cost1 = PERF_MODEL(S1,DL12)
 7:     // cost of executing S2 using the combined layout DL12
 8:     cost2 = PERF_MODEL(S2,DL12)
 9:     // return the sum of the costs
10:     return (cost1 + cost2);
11: end procedure

 1: procedure Perf_model(S1, DL12)
 2:     // classify S1 to memory bound or compute-bound
 3:     T = classify_kernel(S1);
 4:     combinecost ← 0
 5:     // for all field accesses and arrays
 6:     for f ← S1.Fields do
 7:         // find the current layout of f in S1
 8:         D1 = f.getLayout(S1);
 9:         // find the current layout of f in DL12
10:         D2 = DL12.getLayout(f);
11:         if D1 $neq$ D2 then
12:         combinecost+=combine_model(S1.size, D1, D2, T);
13:     return combinecost
14: end procedure
```

In Algorithm 4, we build a $Perf\_model$ function that takes a section $S1$ and a combined data layout $DL12$. It then uses the $combine\_model$ to return the estimated cost. The $combine\_model$ is determined using a set of micro-benchmarks mimicking different kernel characteristics. We classify a kernel into either compute-bound or memory-bound. A kernel is classified statically as compute-bound if the ratio of the compute instruction to the total number of instructions is greater than a threshold (0.6 used in our evaluation), otherwise, it is memory-bound. The $combine\_model$ takes two layouts, the data size (computed similar to Algorithm 3), the memory-bound of the kernel, and returns the performance loss. It is possible that the two affinity graphs cannot be combined due a conflicting affinity value between two fields. In such a case, we use the default layout specified by the programmer.

We wrote a memory-bound micro-benchmark that randomly updates memory locations in a loop inside a kernel. We run this

Table 1: Hardware architectures

Name	Freq	Cores	L1$	L2$
Intel X5660 CPU	2.8GHz	12 (HT)	192KB	1.5MB
NVIDIA M2050 GPU	575 MHz	8	16KB	768KB

micro-benchmark for varying amount of data and different partition sizes. Figure 8 shows the effect of the data layout on a GPU for this memory bound kernel. The x-axis represents the total amount of data being accessed inside the kernel and the y-axis represents the execution time in milliseconds. Each curve represents the execution time for different partition sizes varying from 1 to 12 in this graph. The effect of data layout on a CPU or GPU becomes prominent when the amount of data being accessed increases. We use this curve to determine the *combinecost* in Algorithm 4. In the above formulation, we see that the combine operation happens between sections executed only on the same processor type.

Finally, the formulation requires copy costs which are machine specific. We run micro-benchmarks to determine the cost of moving data from the CPU to the GPU. On integrated CPU+GPU devices, this cost is zero.

5. Experimental Evaluation

We now present experimental results of our implementation of the H2C automatic data layout framework.

5.1 Setup

Table 1 lists the specification of the platform used in our experiments. It consists of an Intel Xeon X5660 CPU with 12 cores running at 2.8GHz and an NVIDIA Tesla with 8 cores running at 575MHz. We used GCC version 4.4.6 with -O2 optimization level to compile our programs.

Table 2 summarizes the benchmarks we use in the evaluation including their compile-time characteristics. Column 1 reports the sources for each benchmark. All benchmarks were originally written in OpenMP and were converted by us to Heterogeneous Habanero-C (H2C) with minimal effort. Typically, all that was required was to change parallel loops to use `forasync` and `finish` constructs. The original layouts for all the benchmarks were SoA. Column 4 reports the number of data-parallel kernels. The Medical Image Registration benchmark consists of 7 compute-intensive data-parallel kernels. The K-Means benchmark has two kernels and the second kernel is executed sequentially in the original OpenMP version of the benchmark as it performs a reduction. Since our current OpenCL code generation does not support reduction, we also implemented this loop as a sequential loop by employing the $s$eq clause in the `forasync` construct. Column 5 reports the number of distinct array or field references accessed in the kernels. Finally, column 6 reports the input data sizes.

For each benchmark, we execute the OpenCL code with the original data layout specified by the programmer and compare it with the automatically generated data layout from our SDL and PDL approaches on the CPU and GPU. We first report results for the SDL approach described in Section 3.3. We denote the SDL results for a benchmark executing only on the CPU as "CPU SDL" and executing only on the GPU as "GPU SDL". We then report results for the PDL approach described in Section 3.2 for only CPU execution(denoted as "CPU PDL") and the combined CPU and GPU execution (denoted as "CPU+GPU PDL").

5.2 SDL Evaluation

We report SDL results for all the data-parallel kernels in our benchmarks. Figure 9 shows the speedups obtained for CPU SDL, GPU SDL for our benchmarks. The CPU SDL is compared with the CPU execution of the same benchmark using the default layout specified by the programmer. Similarly, the GPU SDL is compared with the GPU execution of the same benchmark using the default layout. Table 2 shows the default layouts. We observe performance benefits of up to $2.9\times$ with a geometric mean improvement of $1.4\times$ on the CPU. With the GPU SDL, we found performance improvements of up to $2.2\times$ with a geometric mean benefit of $1.3\times$. 13 out of the total 16 kernels show speedup using our SGML greedy heuristic (Algorithm 1) compared to the baseline layout. It is not surprising to see that many data-parallel kernels show performance improvement from data layout optimization since it results in better cache utilization. The benefits on the GPU can be attributed to the decreased instruction pressure due to the generation of better loads by the NVIDIA backend compiler.

We now discuss the results for each data-parallel kernel: The first seven kernels (Medical-1 to Medical-7) in Figure 9 are from the Medical imaging registration benchmark. Kernels numbered 1, 2, 3, and 7 access $V1, V2, V3$ fields and are grouped together by our SDL algorithm and is named as AoSV layout. Kernels numbered 4, and 5 access fields $U1, U2, U3$ and $V1, V2, V3$ as two groups, but have complementary access patterns (Read, Write). These two groups are kept independently and is named as AoSUV layout. Finally, Kernel 6 accesses the $U1, U2, U3$ fields together and are grouped with the name AoSU layout. We obtain speedup ranging from $1\times$ to $2.4\times$ for all the kernels by using CPU SDL and GPU SDL.

The bars for LBM-1 and LBM-2 in Figure 9 show the speedups obtained for the two kernels in the LBM benchmark. This benchmark has 19 fields for each lattice point in a 3-D space. The first kernel access all the 19 fields for each lattice point while the second kernel access the lattice points for each field. We observe that both kernels require complementary data layouts. The SDL pass combines all the 19 fields in an AoS layout for the first kernel and gives SoA layout for the second kernel. The first kernel gives a speedup of $1.2\times$ on the CPU and $1.7\times$ on the GPU. The second kernel does not benefit from our layout transformation since it uses original baseline SoA layout.

The bars for NBody-1 and NBody-2 in Figure 9 show the speedups obtained for the NBody benchmark. The position and acceleration fields occur together in the first kernel but with different access frequencies. The position, acceleration and velocity fields occur together in the second kernel. The SDL pass groups them into individual groups as AoS layout. This application spends 99% of its execution time in the first kernel. We observe a speedup of $1.4\times$ on the CPU and $1.2\times$ on the GPU for the first kernel.

The bars for KMeans-1 and KMeans-2 in Figure 9 show the speedups obtained for the KMeans benchmark. The first kernel finds the cluster index for each of the input. Hence the SDL pass groups all the clustering features in an AoS layout. We limit the AoS size to 16 which is based on the cache line size. The second kernel is a reduction kernel which is not supported by our current GPU implementation. The first kernel results in a speedup of $2.4\times$ on the CPU and $1.5\times$ on the GPU. The second kernel achieves a speedup of $2.9\times$ on the CPU.

SYR2K kernel reads from the arrays a, and b, and writes to array c. Some accesses to the arrays a, and b are strided. Hence AoS layout benefits from improved cache utilization compared to SoA. We get a speedup of $1.4\times$ on the CPU and $2.2\times$ on the GPU.

GEMVER kernel reads from the fields $u1$, $u2$, $v1$, and $v2$. However the sizes of these fields are very small. Hence the AoS layout performs similar to SoA layout as observed in our combine cost model.

GESUMMV reads from the fields a, b, and x, and writes to fields tmp, y. However the sizes of x and y differ from that of a,

Table 2: Compile-time statistics for the benchmarks used in the evaluation.

Name	Description	Original Layout	Num of Kernels	Num of Fields	Input
`Medical` [9]	Medical Image Registration	SOA	7	6	256×256×256
`LBM` [3, 33]	Computational Fluid Dynamics Simulation	SOA	2	19	300×300×300
`NBody` [16]	Molecular Dynamics	SOA	2	10	10000
`K-Means` [10]	Clustering Algorithm	SOA	2	16	8388608
`GESUMMV` [4]	Linear Algebra Kernel	SOA	1	5	10000
`GEMVER` [4]	Linear Algebra Kernel	SOA	1	9	10240
`SYR2K` [4]	Linear Algebra Kernel	SOA	1	4	2048×2048

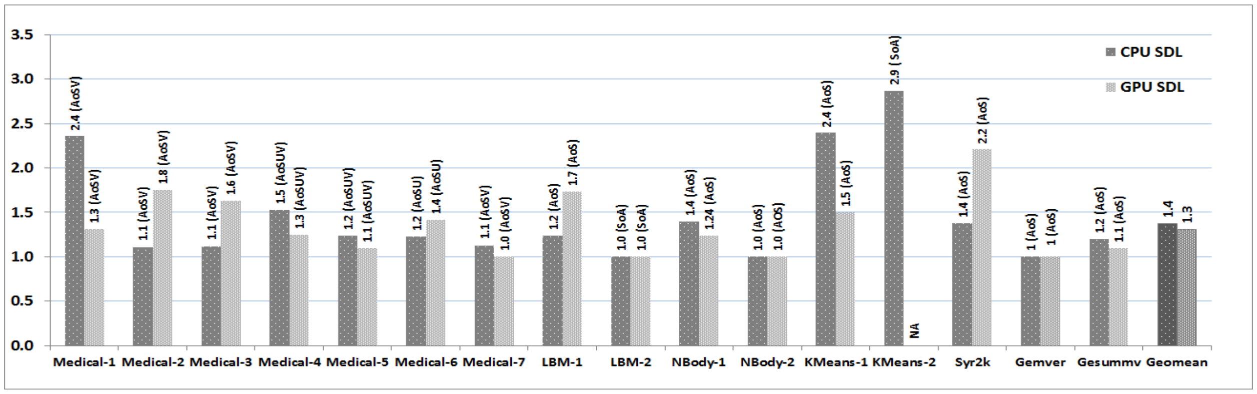


Figure 9: Speedup for all data-parallel kernels on the CPU and GPU by using our SDL algorithm compared to the programmer specified default layout

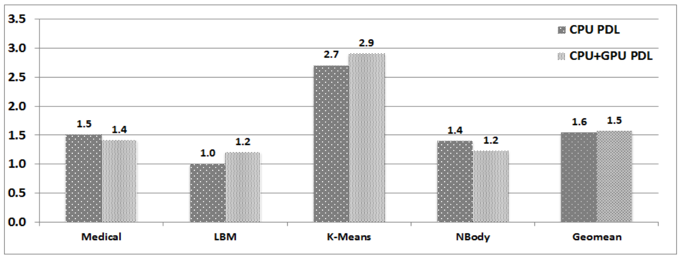


Figure 10: PDL Speedup of the benchmarks

Figure 11: Speedup for multi-kernel benchmarks on the CPU and CPU+GPU by using our PDL algorithm compared to the programmer specified default layout

and b, and hence only a and b are combined by SDL. We observe a speedup of $1.2\times$ on the GPU and $1.1\times$ on the CPU.

5.3 PDL Evaluation

We now report results for our multi-kernel benchmarks: Medical Imaging, LBM, K-Means, and NBody using the PDL. None of the benchmarks have any control flow between the individual data-parallel kernels. We use the combine costs and remap costs as shown in Figure 8 and Figure 7. We can observe from these graphs that the combine cost is $\sim$ 1000 msec between 8-AoS and SoA configurations for the CPU for 1024 MB of data. This means that if a kernel has 8-AoS layout and is combined to an intermediate SoA layout, we estimate the performance loss as $\sim$ 1000 msec. This combine cost is less than the remap cost of $\sim$ 2000 msec (via LDR) on the CPU. On the other hand, the remap cost of $\sim$ 90 msec for LDR is less than the combine cost of $\sim$ 800 msec for 8-AoS and SoA on the GPU for 1024 MB. The copy cost between the CPU and GPU is $\sim$ 2000 msec for 1024 MB. We evaluate the relative performance of PDL on (1) homogeneous multicore CPU and (2) heterogeneous CPU+GPU. Note that the CPU+GPU configuration uses either the CPU or the GPU for a kernel execution, i.e. there is no hybrid execution at the kernel level. However, in a given program, few kernels can run on the CPU and the rest on the GPU. The CPU PDL results are normalized with the multicore CPU execution with the default layout. The CPU+GPU results are normalized with respect to GPU execution using the default layout.

The first three kernels of the medical imaging benchmark have the same layout AoSV as shown in Figure 9. Kernels 4 and 5 have a different layout AoSUV. Now we either have to combine or remap these two layouts. The combine cost between AoSU and AoSUV is 0 because they do not have any common fields. Hence we combine these sections. Similarly, AoSUV and AoSU do not have any common fields and hence we combine kernels 5 and kernel 6. Finally, AoSU and AoSV layouts do not have any common fields and hence kernel 6 and kernel 7 are combined. The overall layout from the PDL pass is AoSUV and a speedup of $1.5\times$ on the CPU and $1.4\times$ on the CPU+GPU. In this case, all the kernels are mapped on the GPU.

LBM is interesting for PDL because both of its kernels prefer complementary layouts as explained in the SDL results. PDL has to decide if it is beneficial to use combine or remap. This benchmark uses a total grid size of approximately 1024 MB. PDL computes the corresponding costs from the combine and remap models described in Sections 4.5 & 4.6. As explained earlier, it is more beneficial to perform combine on the CPU and also to perform remap on the GPU. The combine model for the CPU uses the programmer specified SoA layout because the two kernels cannot be combined. Hence the speedup compared to the baseline layout is $1\times$. On the CPU+GPU, the two kernels are remapped using LDR and we observed a speedup of $1.2\times$. Both the kernels are mapped to the GPU by PDL.

Both kernels in K-Means have been written with a default layout of SoA to enable coalescing on the GPU. That is, all the features of the input data are independent arrays. Both the kernels can take advantage of AoS layout since each kernel is iterating on all the features for every data item. The SDL pass assigned an AoS layout for each of the kernels. As mentioned on the SDL results, the second kernel is executed sequentially on the CPU. In the PDL pass, both kernels will keep the layout as AoS. We observed a speedup of $2.7\times$ on the CPU and $2.9\times$ on the CPU+GPU. The overall speedups obtained are dominated by the speedup from the second kernel which gets executed on the CPU with AoS layout (shown in Figure 9).

The combine cost of the NBody kernels is 0. This is because their corresponding layouts are independent. Hence the PDL output is the same AoS layout. Since the first kernel dominates the majority execution time, the speedup is similar to SDL, which is $1.4\times$ on the CPU and $1.2\times$ on the CPU+GPU.

5.4 Summary

In conclusion, we demonstrate that:

- An automatic SDL transformation that targets a single data-parallel kernel can result in significant performance improvements. On the CPU, we achieve performance benefits of up to $2.9\times$ with a geometric mean improvement of $1.4\times$. On the GPU, we achieve performance improvements of up to $2.2\times$ with a geometric mean benefit of $1.3\times$.
- An automatic PDL transformation that targets multi-kernel programs and kernel mapping can also result in significant performance improvements. On a CPU system, we achieve performance benefits up to $2.7\times$ with a geometric mean improvement of $1.6\times$. On a CPU+GPU system, we achieve performance improvements up to $2.9\times$ with a geometric mean benefit of $1.5\times$.
- Some benchmarks such as LBM do not benefit from PDL transformation on the CPU as its kernels use complementary data layouts.

6. Related Work

Many parallel programming models have emerged that target GPGPU processors, including OpenCL [19], CUDA [5], OpenACC [20], C++AMP [1], Lime [7], Concord [8], etc. To the best of our knowledge, all these past programming systems for GPUs require that data layout be specified by the programmer. Below, we discuss some of the closely related references to this paper.

6.1 Data Layout for CPUs

Ulrich and Kennedy [17, 18, 22, 31] worked extensively on an automatic data alignment and distribution framework for High Performance Fortran. The data layout in their work composed of aligning each dimension of a multidimensional array so as to reduce the cost of communication across different processors. This alignment depends on the access patterns of the array dimensions. They divide a program into *phases*. Each phase consists of a loop nest that covers all the induction variables occurring inside the loop body. They showed that finding the optimal data alignment is an NP-Complete [22] problem in the absence of control flow. Anderson et al. proved that the problem of dynamic remapping in the presence of control flow is NP-hard [6]. Zhong et al. [12] provide a theoretical model for affinity to measure co-accessed data. They also propose a hierarchical reference affinity analysis using k-distance that is used for structure splitting and array regrouping. Our work differs from this past work due to our focus on the data layout and kernel mapping problem for heterogeneous processors.

6.2 Data Layout for Heterogeneous Architectures

Sung et al. [34] use data layout transformation to enable memory level parallelism on structured grid applications for GPUs. Their framework increases the memory level parallelism by distributing the data access by a thread to different banks. Wu et al. [37] propose data reorganization techniques such as data repositioning, duplication, and padding to eliminate non-coalesced access on the GPU. They prove that the problem of data repositioning is NP-hard. DL [15] uses in-place transposition to remap data via cyclic copying. Dymaxion [11] provides an API which is a set of remapping functions from one layout to another. *maprow2col*, *mapdiagonal*, *mapindirect* are some of the mapping functions provided by the API. The remapping of the data is done along with the PCI-E transfer of data. The runtime chunks the data and launches a transformation kernel for each chunk. This allows overlap of remapping and transfer of data. The authors evaluate the performance of hybrid CPU-GPU execution of the k-means application. They use one layout for the CPU and another layout for the GPU with the help of their API. Dymaxion uses a runtime approach which the authors show could be prohibitive. Our automatic data layout framework uses compile-time techniques to change the data layout and leverages the asynchronous features in H2C to reduce the overhead of data remapping. Zhang et al. [38] use a polyhedral framework to determine the optimal data layout for the entire program. Sung et al. [35] design a 3 phase approach to efficiently transpose a matrix in-place on the GPU. The meta-data layout framework [25] in H2C automatically generates a program with the layout specified in a schema file. However, it generates only a single global layout for the entire program based on the meta data specification.

6.3 Kernel Mapping for Heterogeneous Architectures

Qilin [24] provides wrappers for heterogeneous computing and uses adaptive mapping to schedule the work between CPU and GPU. Boyle et al. [28] use machine learning techniques to statically partition the work between CPU and GPU. They execute a suite of benchmarks to build a code feature vector. This feature vector is built using raw kernel features like the number of compute operations, accesses to global memory, accesses to local memory, coalesced memory accesses, average number of data transfers and work-items per kernel. They further derive some combined code features like communication to computation ratio, % coalesced memory accesses, the ratio of local to global memory accesses $\times$ avg. # work-items per kernel, computation-memory ratio.

Compared to past work, our paper provides a two-level formulation to the automatic data layout and kernel mapping problem for CPU+GPU architectures. At the top-level (targeting the entire program), we show that if local data layout for a given kernel is known, the problem is tractable (PTIME) and provide a shortest-path algorithm to compute the best data layout for each data-parallel kernel in the program. We incorporate the cost of copying, remapping and combining data across data-parallel kernels in this formulation. At the bottom-level formulation, we propose a greedy heuristic based on affinity graphs.

7. Conclusion and Future Work

An automatic data layout framework for heterogeneous architectures can dramatically improve programmer productivity and portability in light of current hardware trends. In this work, we propose and implement a two-level hierarchical data layout framework for heterogeneous CPU+GPU architectures. We show that this formulation helps separate kernels running on a CPU and GPU, and it uses an optimal PTIME algorithm to determine the overall data layout given the data layouts for each kernel computed by a greedy search. We provide a reference implementation of the formulation in the Heterogenous Habanero-C compiler framework. The framework uses a parallel intermediate representation to build the affinity graph and a model to estimate the combine and remap costs which are used in determining the overall data layout of the program. Our experimental evaluation shows significant performance benefits of up to 2.7× on a CPU and up to 2.9× on an Intel Xeon CPU+NVIDIA GPU system, which demonstrates the applicability of our approach to both heterogeneous and homogeneous hardware platforms.

In the future, we plan to extend our automatic data layout framework to support dynamic and hybrid scheduling where the data and computation within a single kernel can be split between the CPU and GPU.

References

[1] "C++ Accelerated Massive Parallelism," http://msdn.microsoft.com/en-us/library/vstudio/hh265137.aspx.

[2] "Habanero-C," https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C.

[3] "Halliburton Services," http://www.halliburton.com/en-US/.

[4] "PolyBench/GPU," http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU/.

[5] "The CUDA Specification," 2015, www.nvidia.com.

[6] J. M. Anderson and M. S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," in *In Proceedings Of The SIGPLAN '93 Conference on Programming Language Design And Implementation*, 1993, pp. 112–125.

[7] J. Auerbach and et al., "Lime: a Java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. NY, USA: ACM, 2010, pp. 89–108.

[8] R. Barik, R. Kaleem, D. Majeti, B. Lewis, T. Shpeisman, C. Hu, Y. Ni, and A. Tabatabai, "Efficient Mapping of Irregular C++ Applications to Integrated GPUs," in *International Symposium on Code Generation and Optimization*, ser. CGO'14, Florida, USA, 2014 (To Appear).

[9] Center for Domain Specific Computing, "CDSC Research Applications," 2009, http://www.cdsc.ucla.edu/research/.

[10] Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *In Proceedings of the IEEE International Symposium on Workload Characterization*, ser. ISWC'09, 2009, pp. 44–54.

[11] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: optimizing memory access patterns for heterogeneous systems," in *Proceedings of 11th International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011.

[12] V. De La Luz and M. Kandemir, "Array regrouping and its use in compiling data-intensive, embedded applications," *Computers, IEEE Transactions on*, vol. 53, no. 1, pp. 1–19, Jan 2004.

[13] C. Ding and K. Kennedy, "Inter-array data regrouping," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, L. Carter and J. Ferrante, Eds., vol. 1863. Springer Berlin Heidelberg, 2000, pp. 149–163.

[14] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. NY, USA: ACM, 2009, pp. 152–163.

[15] S. I-Jui, G. Liu, and W.-M. Hwu, "DL: A data layout transformation system for heterogeneous computing," in *In Proceedings of Innovative Parallel Computing*, ser. InPar'12, May, pp. 1–11.

[16] Intel Corporation, "The Intel Threading Building Blocks," 2006, https://www.threadingbuildingblocks.org//.

[17] K. Kennedy and U. Kremer, "Initial Framework for Automatic Data Layout in Fortran D: A Short Update on a Case Study," Rice University, Houston, Texas, USA, Tech. Rep. CRPC-TR93324-S, 1993.

[18] ——, "Automatic Data Layout for High Performance Fortran," in *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, 1995, pp. 76–76.

[19] Khronos, "OpenCL: The open standard for parallel programming of heterogeneous systems," 2010, http://www.khronos.org/opencl/.

[20] Khronos, "The OpenACC: Application Programming Interface," 2011, www.openacc-standard.org/.

[21] J. Knoop and et al., "Lazy Code Motion," in *Programming language design and implementation*, vol. 27. ACM, 1992, pp. 224–234.

[22] U. Kremer, "NP-completeness of Dynamic Remapping," Rice University, Houston, Texas, USA, Tech. Rep. CRPC-TR93330-S, 1993.

[23] X. Liu and et al., "ArrayTool: A Lightweight Profiler to Guide Array Regrouping," in *Proc. of PACT*, 2014, pp. 405–416.

[24] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009.

[25] D. Majeti, R. Barik, J. Zhao, V. Sarkar, and M. Grossman, "Compiler Driven Data Layout Transformation for Heterogeneous Platforms," in *The International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, ser. HeteroPar '13. Aachen, Germany: LNCS, 2013.

[26] G. Mei and H. Tian, "Performance Impact of Data Layout on the GPU-accelerated IDW Interpolation," *CoRR*, vol. abs/1402.4986, 2014.

[27] NVIDIA, "CUDA Toolkit Documentation v6.5," in *NVIDIA Corportation*, 2014. [Online]. Available: http://docs.nvidia.com/cuda/

[28] M. F. P. O'Boyle and et al., "Portable mapping of data parallel programs to OpenCL for heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Washington, DC, USA, 2013, pp. 1–10.

[29] D. Quinlan, "ROSE: Compiler Support For Object-Oriented Frameworks," *Parallel Processing Letters*, vol. 10, pp. 215–226, 2000.

[30] E. Raman and et al., "Structure Layout Optimization for Multithreaded Programs," in *Proc. of CGO*, 2007, pp. 271–282.

[31] B. Robert and et al., "Automatic Data Layout Using 0-1 Integer Programming," in *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1994, pp. 111–122.

[32] V. Sarkar, "Automatic Selection of High-order Transformations in the IBM XL FORTRAN Compilers," *IBM J. Res. Dev.*, vol. 41, no. 3, pp. 233–264, May 1997.

[33] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[34] I.-J. Sung and et al., "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*. NY, USA: ACM, 2010, pp. 513–522.

[35] I.-J. Sung, J. Gómez-Luna, J. M. González-Linares, N. Guil, and W.-M. W. Hwu, "In-place transposition of rectangular matrices on accelerators," in *Proc. of PPoPP*, 2014.

[36] H. Wen-mei and et al., "The Superblock: An effective technique for VLIW and superscalar compilation," *The Jounral of SuperComputing*, vol. 7, pp. 229–248, 1993.

[37] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2013, pp. 57–68.

[38] Y. Zhang, W. Ding, J. Liu, and M. Kandemir, "Optimizing Data Layouts for Parallel Computation on Multicores," in *Proc. PACT*, 2011.

[39] J. Zhao and V. Sarkar, "Intermediate Language Extensions for Parallelism," in *Proc. of SPLASH Workshops*, 2011, pp. 329–340.