Compiler-Driven Data Layout Transformation for Heterogeneous Platforms

Deepak Majeti[1], Rajkishore Barik[2], Jisheng Zhao[1],
Max Grossman[1], and Vivek Sarkar[1]

[1] Rice University
[2] Intel Corporation

Abstract. There is an increasing realization in the community that the layout of data structures in memory can have a significant impact on the performance of sequential and parallel programs. With the advent of heterogeneous platforms (e.g., CPU and GPU integrated on the same die), determining the best data layout can be challenging since the optimal layout for a computational kernel depends on whether the kernel executes on a CPU core, a discrete GPU, or on an integrated GPU (along with other factors). For instance, the GPU memory performance is impacted by upon the number of coalesced memory accesses, whereas CPU memory performance is impacted by factors such as false sharing and data reuse. Since changes in data layout can impact CPU vs. GPU performance, in general, the programmer has to write different versions of CPU and GPU kernels for different architectures and has to select optimal memory layouts for each. This places a severe constraint on code portability.

In this paper, we present a compiler-driven data layout transformation framework for heterogeneous platforms. We introduce a meta-data framework which enables the same source code to be compiled with different data layouts without involving the programmer in the data layout transformations. Our compiler and runtime infrastructure generates efficient code for different architectures based on the meta information. Our experimental results show significant benefits from this approach, and demonstrate that the best data layout for a program is different for CPU vs. GPU execution. On an average, the data layout transformation alone impacted the performance by 7.33× (up to 27.11×) on AMD 4-core A10-5880K CPU, 2.84× (up to 5.57×) on AMD Radeon integrated GPU, 8.32× (up to 29.5×) on NVIDIA Tesla M2050 GPU, 2.19× (up to 5.32×) on Intel 12-core X5660 CPU and 1.9× (up to 3.89×) on Intel integrated i7-3770 GPU for a set of 5 distinct benchmarks.

1 Introduction

Graphics Processing Units (GPUs) have found their way into mainstream computing both in client and server domains. CUDA and OpenCL are the two primary languages targeting these systems for GPGPU programming. Both these languages have been criticized by many researchers [6, 12] as too low-level and device-specific. Writing low-level high-performance CUDA/OpenCL code manually is still cumbersome and error-prone, and thus better suits expert programmers rather than mainstream programmers. This has led to the evolution of many high-level programming models in the last few years that deal with heterogeneity [14, 21, 13, 22, 6, 12]. Although these languages offer significant programmer productivity, they still lag in performance compared to hand-written OpenCL/CUDA programs. In particular, the performance of an application may be sensitive to the organization of the data in the memory on the executing device. We believe that compiler-driven data layout transformations can help bridge this performance gap.

An important aspect of heterogeneous systems is that different devices have different kinds of memory hierarchies. For example, NVIDIA GPUs have L1 and L2 caches that are connected to the system memory via PCIe whereas the integrated GPUs from Intel (e.g., Ivy Bridge and Haswell) have an L3 cache that is connected to the system memory on the same die with a last-level cache (LLC) that is shared between the CPU and GPU. On the host side, the CPU cores have memory hierarchy consisting of L1, L2, L3, and LLC. The varying memory hierarchy on different architectures makes it harder to efficiently map existing data structures of an application on to these different devices since the same data structure may need to be laid out differently on different devices for better performance. The GPU memory performance depends upon the number of coalesced accesses, where as the host CPU memory performance depends on factors such as false sharing and data reuse. Recently, Wu et al [23] have proved that finding the optimal data layout to maximize the number of coalesced accesses on a GPU is NP-complete. Thus, manually writing high performance portable programs or automatically generating efficient code via optimizing compilers without any domain knowledge is challenging given the proliferation of device technologies on heterogeneous architectures and their differing memory hierarchies.

In this paper, we present a meta-data framework that allows both programmers and tuning experts to specify architecture specific and domain specific information for *parallel-for* (*forasync*) loops of a program. A meta-data file is created for an application and is populated with entries on the data layout to be used for a device on the heterogeneous system. The data layout we focus on in this paper include structure-of-array (SOA) and array-of-structure (AOS). Any high level language which has *parallel-for* loops can be extended to accommodate the meta-data framework. In our work, we target the data-parallel **forasync** construct in Habanero-C [2] and integrate our meta-data framework with the Habanero-C compiler and runtime. The code generation for *forasync* construct was extended to generate OpenCL code for targeting heterogenous architectures. The meta-data information is very useful in guiding our compiler optimization passes for the generation of efficient code for a device.

Our paper makes the following contributions:

- A meta-data framework that allows both the programmer and the tuning expert to specify the underlying architecture and domain specific knowledges for parallel-for loops;
- A compiler and runtime framework to automatically generate efficient code based on the meta-data information. We currently focus on AOS-to-SOA and SOA-to-AOS transformations in our compiler;
- An experimental evaluation of our system using a wide variety of heterogeneous architectures which shows the impact of data layout on 5 distinct applications. On an average, the data layout transformation alone impacted

the performance by 7.33× (up to 27.11×) on AMD 4-core A10-5880K CPU, 2.84× (up to 5.57×) on AMD Radeon integrated GPU, 8.32× (up to 29.5×) on NVIDIA Tesla M2050 GPU, 2.19× (up to 5.32×) on Intel 12-core X5660 CPU and 1.9× (up to 3.89×) on Intel integrated i7-3770 GPU.

The rest of this paper is organized as follows. Section 2 presents our meta-data framework. Section 3 discusses the details of our compiler code generation and runtime. Section 4 presents the experimental results on a wide variety of processors. Related work is discussed in Section 5, and finally, Section 6 concludes.

2 Programming Model

Our meta-data framework is built on top of Habanero-C (HC) compiler and runtime infrastructure [9]. The details of the parallel constructs supported by HC can be found at [2]. Our paper focuses on the data parallel *forasync* construct [3]. The syntax of the *forasync* construct is as follows.

```
forasync index(args) size(args) optional<scratchpad(args) seq(args)> {
        // forasync body
}
```

The semantics of the **forasync** construct is similar to a program loop which exhibits *parallel_for* parallelism. The **index** clause is used to specify the loop iterators. The number of variables in the **index** clause gives the dimentionality of the loop. The **size** clause specifies the number of iterations of the loop in each dimension. There are 2 optional clauses, **scratchpad** and **seq** clause. Our language model takes advantage of the different memory regions available on most GPU hardwares with the help of the **scratchpad** and **seq**.

For each host or the device on a heterogeneous system, it is possible to specify the desired data layout for array-based or structure-based data structures of a given *forasync* loop. The data layouts that we focus on are: (1) AOS: array-of-structure; and (2) SOA: structure-of-array. Our compiler (described in Section 3) with the help of the meta-data file is able to transform HC code written in SOA to AOS and vice-versa.

The grammar for the meta-data and an example is shown in Figure 1. The meta-data file consists of a set architecture specific optimization information. The architectural details consist of the data layout information and scratchpad memory allocation information for a given program. Each struct definition has a label *Struct*, a name for the struct and a set of fields. Each field in turn has a label *Field*, the type of the field and the name of the field. The type of fields can be *fp*: a pointer to an array of float values, *dp*: a pointer to an array of double values or *ip*: a pointer to an array of integer values. The scratchpad memory allocation information consists of a set of buffer descriptions. It begins with a label *Scratchpad*, the name of the special memory region, the field, the amount of data which must be cached and the line number of the *forasync*.

---

[3] Our framework is also applicable to other data-parallel programming languages with a parallel-for like construct

arch_name −> Arch **name meta_data** **meta_data** −> (**struct_def**)∗ (**scratchpad_def**)∗ **struct_def** −> Struct **name** (**field_def**)∗ **scratchpad_def** −> Scratchpad **name** (**field_def tile_size line_num**)∗ **field_def** −> Field **type name** length **type** = fp | dp | ip length −> (**digit**)∗ **tile_size** −> (**digit**)∗ **line_num** −> (**digit**)∗ **name** = (**letter**)(**letter**|**digit**)∗ **letter** −> _ |A|B|C| . . . |Z|a|b|c| . . . |z| **digit** −> 1|2|3|4|5|6|7|8|9|0	Arch Intel_GPU **Struct** bodypos **Field fp** posx **Field fp** posy **Field fp** posz **Struct** bodyacc **Field fp** accx **Field fp** accy **Field fp** accz **Scratchpad local Field fp** posx 256 63 **Scratchpad local Field fp** posy 256 63 **Scratchpad local Field fp** posz 256 63 Arch AMD_GPU **Struct** bodypos **Field fp** posx **Field fp** posy **Field fp** posz **Field fp** accx **Field fp** accy **Field fp** accz **Scratchpad local Field fp** accx 1024 63

Fig. 1. Meta-data Grammar (left) and meta-data file Example (right)

Restrictions of our meta-data framework
The user cannot alias the fields specified in the meta-data file. We plan to be resolve this issue with the help of an alias analysis. Another limitation in the programming model is that a variable name cannot be repeated in the whole program in different scopes. This limitation can be avoided by a clever variable renaming mechanism. Also, all fields in a struct must be of the same type. We currently do not support more complex data layouts such as AOSOA (Array-of-structure-of-arrays). We leave this for future work.

3 Implementation

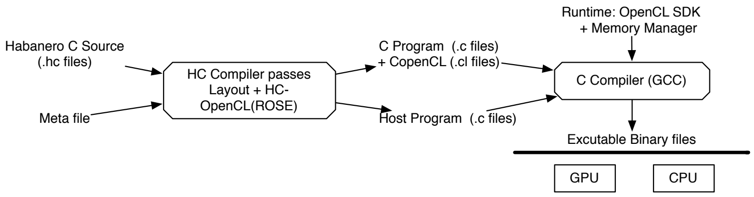


Fig. 2. Compilation Flow

Our overall meta-data framework is shown in Figure 2. The application user writes a program in Habanero-C (HC) using the *forasync* construct. Followed by which, either the developer or the tuning expert specifies the meta-data information for the application. We extend the HC compiler infrastructure to (1) perform data layout transformation based on the meta information; (2) generate OpenCL host and device code. Both these compiler passes are implemented in the ROSE source-to-source compiler framework [17]. The original HC program and the generated OpenCL code are linked together to provide a single executable which runs on the target architecture.

3.1 Data Layout transformation

The compiler pass first parses the specified meta-data file and it creates a meta-data map for each architecture. The mapping is between the fields and the struct name they belong to. The mapping is done for each such struct meta information. If it finds any scratchpad meta information, it records them in the IR.

The data layout transformation (DLT) compiler pass generates the code based on the specified data layout in the meta-data file. It generates code which includes new struct definitions and the code that operates on it. Figure 3 shows the algorithm for transforming the program with a given data layout. `DLT` takes the input program and a meta-data file. `createStructDefinitions(M)` adds the struct definitions as specified in the meta-data file to the AST. These structs are defined only once in the global scope. The DLT pass then iterates over all the functions and performs the steps described in lines 4-7 of Figure 3.

```
1  function DLT()
       Input  : Meta file M and input program P
       Output: Transformed program P'
2      createStructDefinitions(M);
3      for each function F in P do
4          for each formal f in function parameter list do
5              tryAddStructInstances(f);
6          for each instruction I in function body do
7              updateInst(I);
```

Fig. 3. Data Layout Transformation

`tryAddStructInstances(f)` analyzes the function parameters. If any of the parameter names appear in the meta file, an instance of the corresponding struct is declared in the function call. If we abstract the struct as a group of fields names, then one struct instance is declared per group. In next step, `updateInst(I)` checks all pointer or array references in the function body. If any of those reference are via any of the fields in the meta file, then the access is replaced with the corresponding struct instance.

An important factor here is that the type of the function in the original program remains the same. Keeping the function types intact will avoid rewriting the direct and indirect calls to the function.

3.2 Memory Management

In the HC programming model, the programmer allocates heap memory to the fields via standard *malloc* and *calloc* calls. We replace these calls with our specialized memory allocators. We name the allocators, *hc_meta_malloc* or *hc_meta_calloc*. The syntax of the allocators is shown in Figure 4.

```
void *hc_meta_malloc(char *fld_name,size_t num_bytes);
void *hc_meta_calloc(char *fld_name,size_t num_elem,size_t size_elem);
```

Fig. 4. Memory Allocators

hc_meta_malloc or *hc_meta_calloc* are wrappers around the standard *malloc* and *calloc* calls. The allocators also pass in the name of the field to the memory allocator. The field name is required by the memory manager and is explained as follows.

The memory manager handles the different layouts and also creates device buffers. The memory manager has two important components, the memory allocator and the layout handler. During the program initialization phase, the layout

Name	**Description**	**Original Layout**	**Num of Fields**	**Input**
`NBody`	N-Body Simulation	SOA	7	32K nodes
`Medical`	Medical Image Registration	SOA	6	256×256×256
`SRAD`	Speckle Reducing Anisotropic Diffusion	SOA	4	5020 × 4580
`Seismic`	Seismic Wave Simulation	SOA	6	4096 × 4096
`MRIQ`	Matrix Q Computation for 3D Magnetic Resonance Image Reconstruction in Non-Cartesian Space.	SOA	6	64 × 64 × 64

Table 1. Benchmarks

handler reads the meta file and creates a map of the data layout. The memory manager with the help of the field name looks into the layout map and allocates the memory based on the following simple rules.

1. If the field does not belong to any struct layout in the meta file, it means that the programmer wishes it to retain the original layout.
2. If the field belongs to a struct layout group the the allocation happens as follows. Memory is allocated only once per struct group. If memory to the group has already been allocated, then a pointer to the chunk, offset by the field position is returned. If the memory is not allocated to the group, then memory for the whole struct group is allocated. The amount of memory chunk is equal to the number of fields times the number of bytes requested during the memory allocation. Then a pointer to the chunk, offset by the field position is returned.

4 Evaluation

The goal of the experimental evaluation is to prove our metadata framework's ability to extract maximum performance from a given architecture. We compare the impact of data layout on each benchmark on GPUs and multi-core CPUs.

4.1 Experimental Setup

Table 1 describes the benchmarks used in this evaluation. We chose a set of applications whose performance will be most impacted by data layout transformations.

The N-Body particle simulation benchmark was written from scratch for this work. We focus on the compute intensive kernel which calculates the forces between the bodies.

The Medical Imaging benchmark includes kernels from a medical imaging pipeline used to analyze different types of medical images for defects or abnormalities [15]. This application consists of three main phases: *denoising*, *registration*, and *segmentation*. For our evaluation, we focus on the most computationally significant kernel of the three, *registration*.

The SRAD benchmark from the Rodinia benchmark suite [11] is also used. SRAD is used to "remove locally correlated noise" in "ultrasonic and radar imaging applications based on partial differential equations" [18].

The Seismic benchmark suite was created based on the example included in the Intel TBB benchmark suite [4]. Seismic simulates the propagation of waves during seismic activity.

Application	**AOS**	**AOS1**
NBody	**Struct** body **Field fp** posx **Field fp** posy **Field fp** posz **Field fp** accx **Field fp** accy **Field fp** accz	**Struct** pos **Field fp** posx **Field fp** posy **Field fp** posz **Struct** acc **Field fp** accx **Field fp** accy **Field fp** accz
Seismic	**Struct** params **Field fp** S **Field fp** T **Field fp** V **Field fp** D **Field fp** L **Field fp** M	N.A
SRAD	**Struct** direction **Field fp** N **Field fp** S **Field fp** E **Field fp** W	**Struct** direction1 **Field fp** N **Field fp** S **Struct** direction2 **Field fp** E **Field fp** W
Medical	**Struct** disp **Field fp** U1 **Field fp** U2 **Field fp** U3 **Struct** velocity **Field fp** V1 **Field fp** V2 **Field fp** V3	**Struct** disp **Field fp** U1 **Field fp** U2 **Field fp** U3
MRIQ	**Struct** body **Field fp** kx **Field fp** ky **Field fp** kz **Field fp** phiMag	N.A

Table 2. Application meta-data files

Vendor	**Type**	**Model**	**Freq**	**Cores**	**Local Mem**	**L1$**	**L2$**
Intel	CPU	X5660	2.8GHz	12 (HT)	N.A	192KB	1.5MB
Intel	Integrated GPU	i7-3770U	350MHz-1.15GHz	14	64KB (per half-slice)	N.A	N.A
NVIDIA	Discrete GPU	Tesla M2050	575 MHz	8	8x48KB	16KB	768KB
AMD	CPU	A10-5800K	1.4 GHz	4 (HT)	N.A.	16KB	32MB
AMD	Integrated GPU	Radeon HD 7660	800 MHz	6	6x32KB	N.A	4MB

Table 3. Hardware architectures

The MRIQ benchmark from the Parboil benchmark suite [7] computes a Q matrix. The Q matrix represents the scanner configuration used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space. The MRIQ code has been converted to SOA layout by hand.

Table 2 shows the different meta-data files used for each benchmark. Since the default layout is SOA, there is no need of a meta file. All OpenCL kernels, glue code, and different layouts for each of these applications were generated from a HC array-based implementation.

Table 3 lists the hardware architectures used in our evaluation. We use a variety of CPU and GPU systems with differing memory hierarchies in order to demonstrate the benefit of our data layout transformation. The compiler used for the sequential versions of each application GCC 4.4.6 (with the flags `-g -O2`). All OpenCL kernels were compiled with their default optimizations enabled. Intel GPU tests were run using the 2013 Release of the Intel OpenCL SDK [3]. Intel CPU tests were performed using 2011 Release of Intel OpenCL SDK, v1.5 [3]. NVIDIA GPU tests were performed using NVIDIA SDK v5.0 [5]. AMD GPU and GPU tests were performed using AMD APP SDK v2.8 [1].

4.2 CPU and GPU Performance

Figure 5 contains results for all the benchmarks. We compare relative execution time for array and struct data layouts on different CPU and GPU platforms. For a given architecture, we normalize every layout with respect to the fastest executing layout. In this case, **smaller bars imply better performance**. Every column is stacked in 2 levels. The bottom level represents the fraction of total execution time spent on the kernel. This information is retrieved from the OpenCL API. The top stack represents the fraction of total execution time for the remaining execution. This includes communication and OpenCL initialization overheads. The top stack is negligible for Intel GPU because of its integrated GPU and shared memory architecture. As a result, there is no copying overhead. NVIDIA GPU and AMD architectures show copying overheads. For many of the graphs, the AMD GPU exhibits a large amount of overhead. On further inspection, we discovered that the time between OpenCL kernels being submitted to the AMD GPU and being executed by the AMD GPU was significant and accounted for most of this overhead. This could be an implementation error in AMD's OpenCL implementation.

For the N-Body benchmark, we see that the SOA and AOS versions perform similarly on the CPU. Since the number of fields are less, all the loads in an iteration fit into the cache and consecutive iterations do not incur any penalty. The array layout performs better on GPUs because array layout helps in memory coalescing.

For the Seismic kernel, the SOA layout shows better performance on AMD CPU, whereas the AOS layout is better on Intel CPU. This can be attributed to the difference in cache associativity and sizes between AMD and Intel. On the GPU side, array performs well on all 3 GPU hardwares as expected.

The SRAD kernel shows improved performance for the AOS layout relative to the SOA layout for all the architectures. Surprisingly even on the GPU the struct layout performs better than the array layout. This is contrary to GPU best practices. The memory access functions in the SRAD kernel are non-affine and irregular. It is difficult for a compiler or programmer to analyze and determine the right layout. Our framework enables rapid prototyping and testing of different layouts for performance on multiple architectures.

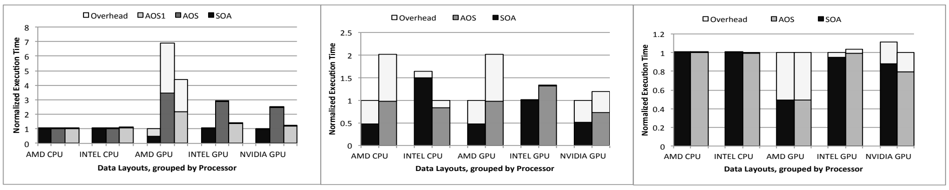
For the MRIQ benchmark, the NVIDIA GPU performs slightly better on the struct layout. For the other architectures, MRIQ exhibits little or no variation across layouts. This is because MRIQ has only 5 fields. For the CPU, if the number of fields loaded are small, then the layout does not make much difference. On the GPU side, the total size of the data was too small to make any real difference.

The medical image benchmark shows some interesting properties for different layouts. The AOS layout is better on the CPU whereas the SOA layout is better on the GPU. Medical image kernel is similar to a 3D Jacobi (stencil) computation. The stencil computation is performed separately on three input buffers and the results are written into corresponding output buffers. Keeping the input buffers in a single struct is helpful for the CPU. This is because when you load a point for one of the stencil, you automatically load the points for the other 2 stencils (multiple points fit in a cache line). The array layout would have caused 3 loads for the same point, one in each of the three stencils. On the GPU side, the array layout is better as expected.

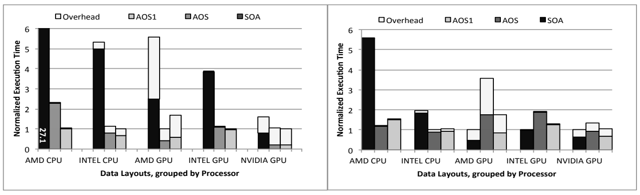
Best practices generally dictate the use of array data layouts on GPUs due to improved coalescence of global memory accesses. However, our SRAD and MRIQ results contradict this knowledge. Our metadata framework enables rapid prototyping and optimization of different data layouts, allowing tuning experts to rapidly discover optimal layouts for complex and irregular applications. For the CPU the layout often depends upon the kernel features and memory access patterns. Our programming model can easily port such applications to different architectures.

5 Related Work

Recently, data layouts have been studied in the context of GPUs. DL [20] uses a mapping function and runtime library support to enable architecture specific data layouts. DL does in-place data marshaling on the GPU. Like DL,



(a) NBody (b) Seismic (c) MRIQ



(d) SRAD (e) Medical Image

Fig. 5. GPU, CPU performance relative to the best layout (Lower the better). The darker bars show results for kernel execution time. The light bar shows results for communication and OpenCL initialization overhead.

Dymaxion [10] proposes a set of index mapping functions which are used to optimize memory mappings, with data marshaling done on the CPU side. Sung et. al. [19] used techniques similar to DL to perform data layout transformations for structured grid applications. Their compiler automatically changes the order of n-dimensional array references to maximize memory access coalescing. With the help of micro-benchmarks, low latency strides and an optimal index map are discovered. This technique requires manual host code changes. The main disadvantage of the techniques listed in this paragraph is that the overhead of runtime data marshaling can eliminate or reduce the performance benefits of optimal data layouts. Our compiler based approach does not incur this added overhead. The above runtime-based techniques are also either restricted to a class of applications (such as grid applications) or require manual changes by the programmer. Our compiler-based approach does not require any manual coding.

Ren et. al. [8] introduce an interpreter-based SIMDization mechanism that can parallelize the sequential programs that traverse on irregular data structures (e.g., binary decision tree and regular expression matching). To reduce memory latency, they chose different compacting policies based on layout including depth-first, breadth-first, and level-by-level. These policies improve memory access latencies for the irregular data structures. Compared to their work, our meta-data framework is not tied to any specific applications and that it can be easily extended to support the above data-layouts.

Liu et al [16] describe an automatic layout transformation that first divides arrays into blocks and then maps them to processing units with minimal overlap. Their approach does not perform any AOS to SOA transformation.

6 Conclusions

We present a compiler-driven data layout transformation that is applicable to any data parallel parallel_for programming model. The data layout transformation uses a "meta-file" approach which enables the same source code to be compiled with different layouts without involving the programmer worrying about it. Our experimental results show significant benefits from this approach and demonstrates that the best data layout for a given program can be different for CPU vs. GPU execution. With the growing users of GPUs in mainstream computing, it is important to have a tool like ours to understand the performance debugging. In future, we would like to develop a general heuristic to automatically perform data layout transformation that works across all platforms.

References

1. Amd app sdk v2.8. `http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk`.
2. Habanero-c. `https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C`.
3. Intel opencl sdk. `http://software.intel.com/en-us/vcsource/tools/opencl-sdk`.
4. Intel thread building blocks. `http://threadingbuildingblocks.org/`.
5. Nvidia sdk. `https://developer.nvidia.com`.
6. Openacc. `http://www.openacc-standard.org`.
7. Parboil benchmark suite. `http://impact.crhc.illinois.edu/parboil.aspx`.
8. *SIMD Parallelization of Applications that Traverse Irregular Data Structures*. IEEE Computer Society, 2013.
9. Chatterjee et al. Integrating asynchronous task parallelism with mpi. IPDPS'13.
10. Che et al. Dymaxion: optimizing memory access patterns for heterogeneous systems. SC '11, pages 13:1–13:11, New York, NY, USA, 2011. ACM.
11. Che et al. Rodinia: A benchmark suite for heterogeneous computing. ISWC'09, pages 44–54, Oct 2009.
12. Microsoft Corporation. C++ accelerated massive parallelism specification. `http://msdn.microsoft.com/en-us/library/vstudio/hh265136.aspx`.
13. Dave Cunningham, Rajesh Bordawekar, and Vijay Saraswat. Gpu programming in a high level language: compiling x10 to cuda. X10 '11, pages 8:1–8:10, New York, NY, USA, 2011. ACM.
14. Dubach et al. Compiling a high-level language for gpus: (via language support for architectures and compilers). PLDI '12, pages 1–12, NY, USA, 2012. ACM.
15. Center for Domain Specific Computing. Cdsc research applications.
16. Liu et al. Data layout optimization for gpgpu architectures. PPoPP '13, pages 283–284, NY, USA, 2013. ACM.
17. LLNL. Rose compiler infrastructure. `http://rosecompiler.org/`.
18. Rodinia Benchmark Suite. Srad wiki page.
19. Sung et al. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. PACT '10, pages 513–522, New York, NY, USA, 2010. ACM.
20. I-Jui Sung, G.D. Liu, and W.-M.W. Hwu. Dl: A data layout transformation system for heterogeneous computing. InPar'12, pages 1–11, May.
21. Didem Unat, Xing Cai, and Scott B. Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. ICS '11, pages 214–224, New York, NY, USA, 2011. ACM.
22. Cave Vincent, Zhao Jisheng, Shirako Jun, and Sarkar Vivek. Habanero-java: the new adventures of old x10. PPPJ'11, 2011.
23. Wu et al. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. PPoPP '13, pages 57–68, New York, NY, USA, 2013. ACM.