Automatic Data Allocation, Buffer Management and Data Movement for Multi-GPU Machines

A Thesis

Submitted For the Degree of

Master of Science (Engineering)

in the Computer Science and Engineering

by

Thejas Ramashekar

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

OCTOBER 2013

Acknowledgements

I take this opportunity to thank all those who made this thesis work possible. First and foremost, I thank my advisor Dr. Uday Bondhugula for his invaluable guidance. He helped us to get started by guiding us thoroughly through our first joint work, out of which I found the motivation and the necessary expertise for the main problem solved in the thesis. He gave me enormous amount of freedom in selecting the problem and approaching the solution and was always there with a timely guidance when I needed it.

I specially thank the members of the Multi-core Computing Lab who made the day-to-day environment both academically and otherwise refreshing. Without them, the lab would not have been the wonderful workplace that it is. Special thanks to Roshan Dathathri for being always enthusiastically available for a technical discussion and his help towards reviews and suggestions. I would like to thank my friend Jay Thakkar for all his help with Tikz diagrams used in this thesis.

I would also like to specially thank my wife Pallavi for her constant support during times of need, sharing in my happiness during times of joy, and timely warnings whenever I slacked in my work.

I have been fortunate to have a enviable group of close friends whose support and company made the entire experience a truly memorable one.

Finally, I would like to thank my parents, and sister for their support and encouragement without which I would not have even thought of pursuing higher education.

Publications based on this Thesis

1. Thejas Ramashekar, Uday Bondhugula, Automatic Data Allocation and Buffer Management for Multi-GPU machines. In the ACM Transactions on Architecture and Code Optimization, Vol. 10, No. 4, Article 60, Publication date: December 2013. Selected for presentation at HiPEAC 2014, Jan 2014, Vienna, Austria.

2. Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In the 22nd International Conference on Parallel Architectures and Compilation Techniques (ACM/IEEE PACT), September 2013, Edinburgh, Scotland.

Abstract

Multi-GPU machines are being increasingly used in high performance computing. These machines are being used both as standalone workstations to run computations on medium to large data sizes (tens of gigabytes) and as a node in a CPU-MultiGPU cluster handling very large data sizes (hundreds of gigabytes to a few terabytes). Each GPU in such a machine has its own memory and does not share the address space either with the host CPU or other GPUs. Hence, applications utilizing multiple GPUs have to manually allocate and manage data on each GPU.

A significant body of scientific applications that utilize multi-GPU machines contain computations inside affine loop nests, i.e., loop nests that have affine bounds and affine array access functions. These include stencils, linear-algebra kernels, dynamic programming codes and data-mining applications. Data allocation, buffer management, and coherency handling are critical steps that need to be performed to run affine applications on multi-GPU machines. Existing works that propose to automate these steps have limitations and inefficiencies in terms of allocation sizes, exploiting reuse, transfer costs and scalability. An automatic multi-GPU memory manager that can overcome these limitations and enable applications to achieve scalable performance is highly desired.

One technique that has been used in certain memory management contexts in the literature is that of *bounding boxes*. The bounding box of an array, for a given tile, is the smallest hyper-rectangle that encapsulates all the array elements accessed by that tile. In this thesis, we exploit the potential of bounding boxes for memory management far beyond their current usage in the literature.

In this thesis, we propose a scalable and fully automatic data allocation and buffer

management scheme for affine loop nests on multi-GPU machines. We call it the Bounding Box based Memory Manager (BBMM). BBMM is a compiler-assisted runtime memory manager. At compile time, it uses static analysis techniques to identify a set of bounding boxes accessed by a computation tile. At runtime, it uses the bounding box set operations such as union, intersection, difference, finding subset and superset relation to compute a set of *disjoint* bounding boxes from the set of bounding boxes identified at compile time. It also exploits the architectural capability provided by GPUs to perform fast transfers of rectangular (strided) regions of memory and hence performs all data transfers in terms of bounding boxes. BBMM uses these techniques to automatically allocate, and manage data required by applications (suitably tiled and parallelized for GPUs). This allows it to (1) allocate only as much data (or close to) as is required by computations running on each GPU, (2) efficiently track buffer allocations and hence, maximize data reuse across tiles and minimize the data transfer overhead, (3) and as a result, enable applications to maximize the utilization of the combined memory on multi-GPU machines. BBMM can work with any choice of parallelizing transformations, computation placement, and scheduling schemes, whether static or dynamic. Experiments run on a system with four GPUs with various scientific programs showed that BBMM is able to reduce data allocations on each GPU by up to 75% compared to current allocation schemes, yield at least 88% of the performance of hand-optimized OpenCL codes and allows excellent weak scaling.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 GPUs in high performance computing

In the nineties, the use of GPU was mainly restricted to graphics processing. The processor cores and memory inside GPUs were all designed and optimized to be good at pixel shading – so much so that the cores were termed as *shaders*. All the processing elements executed the same instruction in parallel on an independent pixel element in a Single Instruction Multiple Data (SIMD) manner. Since 2000s, researchers have found ways to utilize the massively parallel power of GPUs for general purpose scientific computations [15] and in the last decade this has gained significant traction. To be suitable for general purpose computations, GPUs have undergone significant evolution in terms of both hardware and the associated software infrastructure. Such GPUs are commonly referred to as General Purpose GPUs (GPGPUs). Today GPGPUs have become a key component in High Performance Computing (HPC) setups. In the June 2013 list of the top 500 supercomputers, 16 of the top 100 supercomputers contain both CPU and GPU processors [40]. This has led to an enormous amount of research and development efforts towards a software ecosystem (mainly compilers and runtime libraries) for GPGPUs, that can achieve better performance, improve their ease of use, and integrate them with the CPU as a seamless heterogeneous system. From now on, we just use the term GPU to refer to GPGPU.

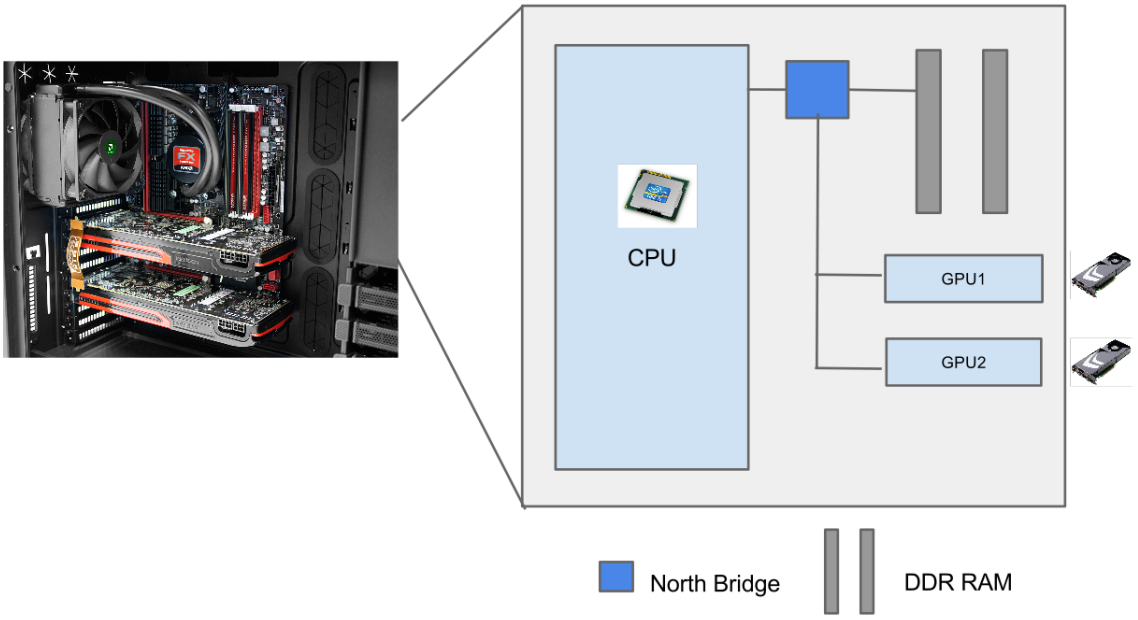


Figure 1.1: Multi-GPU machine setup (Photo Courtesy: AMD)

1.2 Towards Multi-GPU machines

Once the benefits of using GPUs became clear, scientists and researchers started using more than one GPU on a single machine to further increase its computing capacity. Today, multi-GPU machines are becoming commonplace in HPC setups. These machines are being used both as standalone workstations to run computations on medium to large data sizes (tens of gigabytes) and as a node in a CPU-MultiGPU cluster handling very large data sizes (hundreds of gigabytes to a few terabytes).

A schematic diagram of a multi-GPU machine is shown in Figure 1.1. It consists of a host CPU and one or more GPUs connected on the PCIex slots. The CPU memory (DDR RAM) is connected through the memory controller (North bridge). The PCIex bus that interconnects the GPUs also connects to the north bridge. However, each of the GPUs have their own memory, and they do not share the address space with either the CPU or with other GPUs. Hence, all the data needed by the computations has to be explicitly copied from the CPU onto each GPU before computations are run and

once the computations complete, the results have to be explicitly copied back. Whenever the computations on a GPU update their own memory, the updations have to be synchronized across all the GPUs through inter-device data transfers in order to satisfy the program dependences.

1.3 Programming challenges on a multi-GPU machine

Programming a multi-GPU machine to efficiently utilize its combined power is not straight forward. Programmers have to address multiple non-trivial challenges. In this section we briefly review the important ones.

1.3.1 Computation partitioning and load balancing

In order to utilize the combined processing power of all the GPUs, the computations have to be partitioned into tasks and each task has to be scheduled on the available GPUs. The data required for these computations has to be explicitly copied to each GPU. Also, to ensure that the compute resources are optimally utilized, the partitioning has to be load balanced according to the capability of each GPU.

1.3.2 Data allocation and buffer management

Once the tasks have been distributed onto the GPUs, the data required by a task have to be allocated on the GPU on which it runs. GPUs have limited memory – typically 3 to 6 GB. Hence, applications working on data sizes larger than this size cannot allocate their entire data on a single GPU. In a multi-GPU machine, even if the combined memory size of the GPUs is larger than the data size, utilizing them to work on data sizes proportional to the combined memory size is currently not an easy task. Also, inorder to maximize the utilization of the available GPU memory the buffers already allocated on the GPU have to be tracked and reused so that there is no redundant allocation and the associated

data transfers.

1.3.3 Inter-GPU data movement

The GPUs within a node are physically connected on the PCIex bus which has a peak unidirectional and bidirectional bandwidth of 8 GB/s and 16 GB/s respectively. Because of this, inter-GPU data movement becomes the bottleneck when the amount of data transferred between devices is significant. Inefficient data movement schemes can result in cost overhead which overcomes the benefit of computation distribution onto multiple GPUs. Hence minimizing the data movement between GPUs is of paramount importance to minimize the overall execution time. Also, for many programs even minimal data movement volume has significant overhead. One can instead try to hide the overhead by overlapping data movement within the computation time. To achieve a high degree of scalability, this overlap has to be maximized such that all the data movement complete while the computations are still going on.

1.3.4 Existing approaches

Traditionally these challenges have been tackled using manual and application specific techniques. This generally involves an expert programmer manually partitioning the computation and/or data, after studying the nature of the problem and distributing these partitions to the available compute devices. This effort is tedious, error prone and time consuming. To ease the effort involved in efficient programming of single/multi-GPU setups, current and past research has tried to automatically handle one or more of the sub-challenges. Some works [4, 5, 23, 24, 41] propose automatic code generation techniques that generate GPU code usually in terms of CUDA [11] or OpenCL [27]. Other works [3, 18, 19, 21, 35, 36] propose runtime techniques to handle task placement and scheduling, data allocation and inter-device data movement. Recent proposals like OpenACC [26] and some proprietary compilers [34, 42] target ease of programming through directive based approach similar to OpenMP. They provide a set of computation

and data directives which can be used to annotate the application. The compiler then automatically generates the GPU code and code for the necessary data transfers.

1.4 Affine loop nests

Affine loop nests are loop nests that have affine bounds and the array access functions in the computation statements are affine. A significant body of scientific applications that utilize multi-GPU machines contain computations inside affine loop nests. These include stencils, linear-algebra kernels, dynamic programming codes and data-mining applications. The work presented in this thesis is applicable to all of these.

1.4.1 Running affine loop nests on multi-GPU machine

To run affine loop nests with large datasets on a multi-GPU machine and achieve scalable performance, one has to perform the following steps efficiently:

1. Break up the loop computations into smaller, parallel task units called *tiles* and distribute them across multiple GPUs.
2. Allocate the array data required by each tile on the GPU on which the tile executes.
3. Perform the computations on each GPU in parallel.
4. Once the computations are run, perform the data transfers which are required to maintain coherency between GPUs.
5. Aggregate the final results from the GPUs onto the host CPU.

1.5 Need for a multi-GPU memory manager

Data allocation, buffer management, and coherency handling is a critical part of the above steps. From this aspect, currently, applications utilizing multiple GPUs resort to manual programming efforts which are often tedious, time consuming and error-prone.

Existing works in this area are either manual, application-specific techniques [3, 11, 26], or automatic schemes [21] that have limitations and inefficiencies in terms of allocation sizes, reuse exploitation, coherency costs and scalability. Hence, an automatic memory management framework for multi-GPU machines that can overcome these shortcomings and enable applications to achieve scalable performance is much needed. In this thesis, we present the design of such an automatic multi-GPU memory manager that embeds itself into steps (2), (4) and (5) above, and performs those tasks efficiently on behalf of the application.

1.5.1 Desired capabilities

An important metric in measuring the effectiveness of a multi-GPU memory manager is its ability to allow applications to maximize the utilization of GPU memory. It should enable applications to work with large dataset sizes proportional to the combined GPU memory size without any loss in performance. We will refer to this scaling requirement as *data scaling*. Data scaling can be seen as a form of weak scaling but has an emphasis on data size (memory utilization) rather than on the workload (computation). This notion of weak scaling is more precise in the context of our work as we will see.

In order to achieve high degree of data scaling, an automatic memory manager should have the following abilities:

- To identify and minimize data allocation sizes for a tile such that the data required by a tile fits within individual GPU memory. Any scheme chosen to achieve this, must also ensure that the cost of accessing the data from a smaller buffer is not significant.
- To identify the data already present on a GPU and reuse it across tiles, thereby minimizing redundant allocations and CPU-GPU data movement costs.
- To keep the data transfers minimal and efficient to reduce overhead.
- To ensure that the overhead of achieving the above tasks do not adversely affect overall execution time of the program.

1.6 Bounding Box based Memory Manager

One technique that has been used in certain memory management contexts in the literature is that of *bounding boxes* [4, 21]. Bounding box of an array, for a given tile, is the smallest hyper-rectangle that encapsulates all the array elements accessed by that tile. Bounding boxes have been mainly used due to the simplicity of accessing the elements in them. In this thesis, we exploit the potential of bounding boxes for memory management far beyond their current usage in the literature, while providing the following key insights:

- Bounding boxes can be subjected to standard set operations like union, intersection, difference etc, at *runtime* merely by performing simple checks and arithmetic on their vertices.
- GPUs are architecturally designed to be efficient at copying rectangular regions of memory.

With the above insights in mind, we propose the Bounding Box based Memory Manager (BBMM). BBMM is a scalable and fully automatic compiler-assisted runtime memory manager for multi-GPU systems. At compile time, it uses static analysis techniques to identify a set of bounding boxes accessed by a computation tile. At runtime, BBMM uses the bounding box set operations to compute a set of *disjoint* bounding boxes from the set of bounding boxes identified at compile time. This reduces unnecessary and redundant memory allocations. These disjoint bounding boxes are then tracked on a per GPU basis that allows it to maximize inter-tile data reuse and minimize CPU-GPU data movement overhead. All data transfers are performed in terms of bounding boxes thereby exploiting the architectural benefits provided by the GPUs. The runtime operations incur negligible overhead.

1.7 Contributions

The main contributions of this thesis are as follows:

- We present the design of a fully automatic, efficient, and highly scalable memory manager for affine loop nests on multi-GPU machines.
- We present a compiler-assisted runtime algorithm to store and manage accessed array data as a set of *disjoint* bounding boxes.
- We present a runtime buffer management scheme that allows applications to (1) maximize inter-tile data reuse thereby maximizing memory utilization and minimizing CPU-GPU data movements, (2) work with data sizes greater than the combined memory size of the GPUs.
- We present an efficient inter-GPU data movement technique that (1) minimizes the volume of data moved due to flow dependences (2) performs all copies efficiently in terms of bounding boxes by exploiting the architectural capability provided by the GPUs (3) maximizes compute-copy overlap and helps to further reduce the data movement overhead.
- Experiments on a system with four GPUs and large dataset sizes showed that, BBMM was able to achieve reduction in data allocation sizes of up to 75% compared to current allocation schemes, achieve at least 88% of the performance of hand-optimized code, and achieve excellent weak scaling.

The rest of this thesis is organized as follows. Chapter 2 gives examples to motivate the important techniques proposed in the thesis. Chapter 3 presents the necessary background material. Chapter 4 is the central chapter of this thesis. It describes the data allocation, buffer management and data movement schemes of BBMM along with the implementation details, experimental results and analysis. Chapter 5 gives more details about the data movement technique used in BBMM and presents the detailed results. It also describes a new technique for maximizing compute-copy overlap and presents the results for the same. Chapter 6 discusses related work and Chapter 7 provides conclusions and future work.

Chapter 2

Motivating Example

In this chapter, we present motivating examples that gives the readers a more concrete view of the ideas introduced in the previous chapter. We first provide a brief discussion of the general structure of the code that runs on a multi-GPU machine. We then take Floyd-Warshall algorithm as an example program and illustrate various contributions of this work such as the data allocation scheme, the inter-GPU coherency scheme, and the benefits of inter-tile data reuse. We also motivate the readers to the benefits of our techniques by providing comparative numbers on a sample tile size.

2.1 General structure of affine programs running on a multi-GPU machine

An affine program can consist of one or more arbitrarily nested affine loop nests. For an affine loop nest to be suitable to run on a GPU setup, it has to have at least one parallel loop dimension (preferably a parallel band of two or three loops). The parallel loop band can be surrounded by zero or more outer serial loops. Inside the parallel band, there can be more loop dimensions. Each iteration of the parallel band can be executed independently on a GPU thread. The parallel band can be tiled to appropriate size based on the GPU capabilities.

Figure 2.1 illustrates the general structure of a single affine loop nest for a multi-GPU machine. In a program with multiple independent affine loop nests, this structure is repeated for each of them. The execution begins on the host CPU with the outermost serial loop (if it exists) as shown in line 11. For each surrounding serial loop iteration, the tiles of the parallel band are distributed among the available GPUs (line 13). For each tile assigned to a GPU, the data allocation function is called to allocate the necessary data for the tile (line 4). This function is in turn implemented by a memory manager which performs the actual allocations based on its internal schemes. Once the data is ready, the computation kernel is launched. After a tile finishes computation, explicit inter-GPU data movement has to be performed (henceforth referred to as coherency) to ensure that data allocated across GPUs are in sync before the next serial iteration (line 6). This involves two distinct data transfers. The coherency data has to be first copied out of the source GPU that updated the data onto the host CPU. We call it the *flow-out* transfer (described later in Section 4.5). This data has to be then updated onto the GPUs that need it in the subsequent serial iterations. This is called the *flow-in* transfer. Implementing the coherency scheme efficiently is again the job of the memory manager. At the end of all iterations, the result is aggregated from all GPUs onto the host CPU.

2.2 Floyd-Warshall algorithm

The `floyd-warshall` algorithm computes the shortest-path between every pair of vertices in a directed weighted graph. The input to the algorithm is a path matrix, which is initialized to the cost of edges between a pair of vertices if it exists and infinity otherwise. In each iteration, the algorithm finds the shortest path between any two vertices, *passing through* a pivot vertex considered in that iteration. This is computed as the minimum value of the current path weight, and the sum of the path weights of the source to pivot and pivot to the destination. Figure 2.2 shows the code for `floyd-warshall`. It has an outer serial loop k and inner parallel loops i and j. It has a single `path` array and three distinct access functions, `path[i][j], path[i][k],`

```
1  void gpu_mgmt_thread(Device * gpu)
2  {
3    for(tid=gpu->stile;tid<=gpu->etile;tid++) {
4      allocate_data(gpu, tid);
5      launch_kernel(gpu, tid);
6      perform_coherency(gpu, tid);
7    }
8  }
9  int main()
10 {
11   for(ser=0; ser < NUM_ITER; ser++) {
12     for(i=0;i<NUM_GPUS;i++) {
13       distribute_parallel_loop(i,&gpu[i].stile,&gpu[i].etile);
14       spawn_thread(&gpu[i], gpu_mgmt_thread);
15     }
16     synchronize_mgmt_threads();
17   }
18   aggregate_results();
19   ...
20   // more affine loop nests can follow with the same
         structure
21 }
```

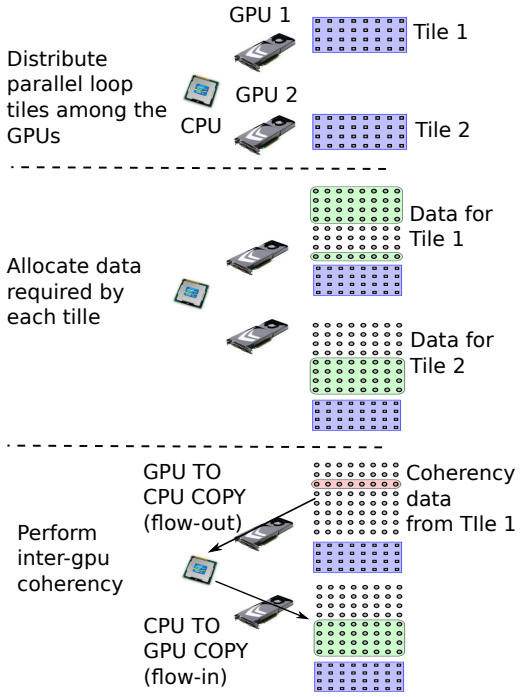


Figure 2.1: General structure of an affine loop nest for a multi-GPU machine

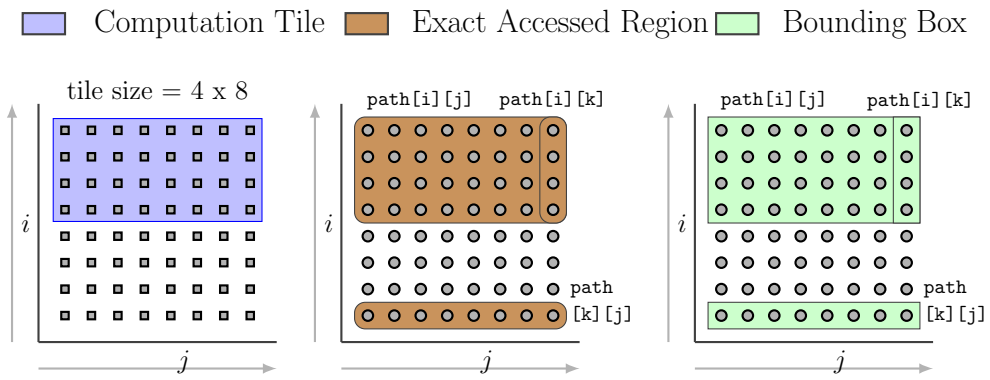
`path[k][j]`. `floyd-warshall` has non-uniform array access patterns. Hence, depending on the value of k, these access functions access regions of array that either intersect or are disjoint. Figure 2.3a shows the iteration space for floyd-warshall with N set to 8 and k set to 7. A tile of size 4 x 8 is highlighted. When k is 7, the regions of array accessed by this tile with `path[i][j]` and `path[i][k]` overlap, whereas the region accessed with `path[k][j]` is disjoint from the other two (Figure 2.3b).

2.2.1 Per-tile data allocation

To allocate data for the tile, BBMM first identifies the regions of `path` array accessed by this tile in terms of bounding boxes (called the *initial bounding boxes*) as shown in Figure 2.3c. Some works in the literature [4, 21] propose to allocate the bounding box over the convex hull of the accessed array regions. Such a scheme would end up allocating the entire array as shown in Figure 2.3d, even though the actual region accessed by the tile is much smaller. BBMM however, performs set operations such as union, intersection,

```
for (k=0; k<N; k++) /* outer serial loop */
  for (i=0; i<N; i++) /* outer most parallel loop */
    for (j=0; j<N; j++)
      path[i][j]=((path[i][k]+path[k][j])<path[i][j])?path[i][k]
                  +path[k][j]:path[i][j];
```

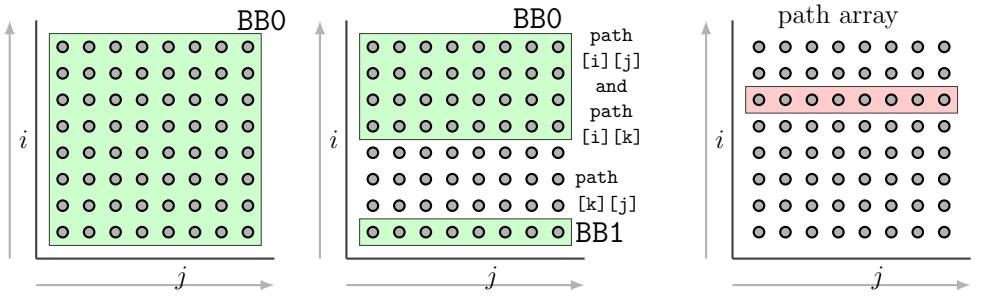
Figure 2.2: Floyd-Warshall code



(a) Iteration space of a tile (size: 4 x 8, k=7)

(b) Exact array regions accessed by the tile

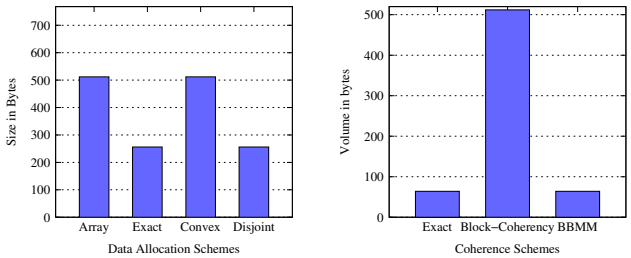
(c) Initial bounding boxes



(d) Convex bounding box

(e) Disjoint bounding boxes

(f) Coherency bounding box (N=8,k=1)



(g) Per-tile data allocation size

(h) Per-iteration coherency volume

Figure 2.3: Per-tile data allocation, coherency and reuse exploitation on `floyd-warshall`

difference etc on these initial bounding boxes and allocates the data required by the tile in terms of *disjoint* bounding boxes. Figure 2.3e shows the two disjoint bounding boxes for the tile – one for `path[i][j] and path[i][k]` combined and another disjoint one for `path[k][j]`. The combined size of the disjoint bounding boxes is much smaller than the bounding box for the convex one as shown in Figure 2.3g and equal to the exact array regions accessed by the tile.

2.2.2 Inter-GPU coherency

Dependence analysis of `floyd-warshall` can show that, in the k^{th} iteration, a GPU reads the elements of the k^{th} row, which would have been updated possibly by another GPU, in $k-1^{th}$ iteration. Hence to maintain coherency, at the end of $k-1^{th}$ iteration, this row has to be transferred from the source GPU that updated it to all other GPUs. For example, when k is 1, the row highlighted in Figure 2.3f needs to be synced with other GPUs. However, the coherency schemes used in the closely related works [21] transfer the entire data allocation block which contain at least one updated element (henceforth referred to as *block-coherency* scheme). In our example, this is the bounding box of `path[i][j]` in the best case (which by itself is much larger than the actual amount of data that need to be synchronized), and almost the entire array in the worst case i.e, when k is 6. Figure 2.3h shows the comparison of data movement volume due to coherency considering only the best case. Even in the best case, there is a difference of $8\times$ between the block-coherency scheme and the exact volume required.

2.2.3 Exploiting inter-tile reuse

For values of k between 0 and 3, the bounding box required by `path[k][j]` is already allocated on the GPU as part of the bounding box for `path[i][j]` (allocated when k was 0). BBMM can very easily detect such data which is already present on the GPU using the subset and superset relationship on bounding boxes. As we show later in the results, many programs perform significantly better with reuse exploitation.

Chapter 3

Background

In this chapter, we briefly review the background required for the understanding of the work presented in the thesis. Section 3.1 provides an overview of the GPU architecture. Section 3.2 provides an overview of OpenCL along with a sample program. Section 3.3 provides a brief overview of the Polyhedral framework used in some of the techniques presented in the thesis.

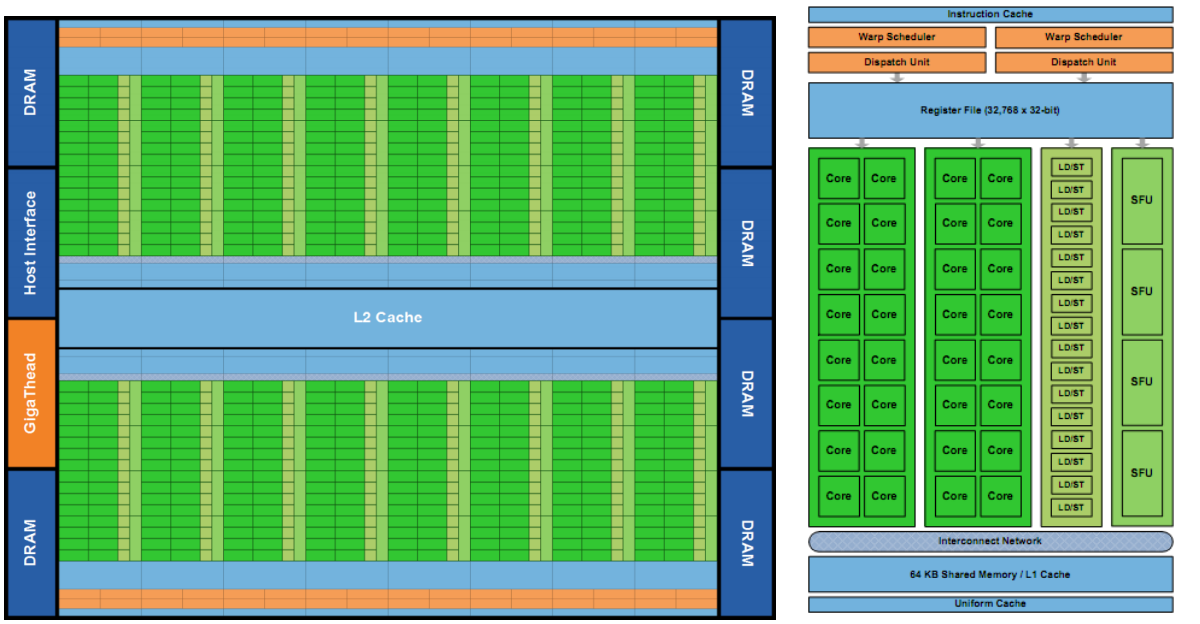
3.1 Overview of GPU architecture

In this section, we briefly give an overview of GPU architecture by taking NVIDIA's *fermi* [13] as the example. Though different GPUs differ slightly in capabilities depending on the workload and cost, the underlying hardware elements do not differ significantly.

3.1.1 NVIDIA Fermi GPU architecture

Compute architecture

Fermi consists of a set of 16 streaming multiprocessors (SM), with each SM containing 32 CUDA cores making a total of 512 cores. Each CUDA core consists of a register file with 32,768 32-bit registers, a fully pipelined integer and floating point unit. The floating point unit supports the fused multiply and add (FMA) for both single and double



(a) Fermi overall architecture (b) Fermi streaming multiprocessor

Figure 3.1: Fermi Architecture (Courtesy:NVIDIA)

precision arithmetic. Each SM also consists of 16 load/store units and 4 special function units for executing instructions like sine and cosine. The SM schedules threads in groups of 32 parallel threads called *warps*. Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently.

Memory architecture

Fermi has six 64-bit DRAM memory modules, connected through a 384-bit memory interface. This can support up to 6 GB of global memory that is visible across all SMs. Each SM also has 64 KB of on-chip memory that can be configured either as 48 KB of Shared memory with 16 KB of L1 cache or as 16 KB of Shared memory with 48 KB of L1 cache. It also has a 768 KB of the L2 cache which is shared across all the SMs. GPU memories generally have high bandwidth, but significant latency. Hence, the optimal utilization of memory bandwidth can occur when the accesses to the memory are *coalesced*; i.e, adjacent threads in a warp access adjacent elements in memory.

Code execution and data availability

The current GPU architectures including the Fermi, can only work as a co-processor for a host CPU. The GPU itself cannot initiate a program execution. The code that runs on the GPU is referred to as the *kernel*. The host CPU is responsible for scheduling the execution of a kernel on the GPU. Also, GPUs do not share the address space with the host CPU. Hence, the data required by a kernel has to be explicitly copied from the CPU's memory to GPU's memory. Once the kernel completes execution, the responsibility of copying the results from GPU to CPU rests with the CPU.

Synchronization and consistency

GPUs provide the following two methods of synchronization.

- Threads within a SM can synchronize using barrier synchronization primitives.
- Threads across SMs can synchronize by implementing synchronization primitives such as mutexes, with the help of atomic operations such as atomic compare and swap, provided by the GPU.

GPUs only guarantee a relaxed consistency model [16]. Threads within a single SM can share data efficiently using the fast barrier synchronization primitives. However, for threads across SMs, the changes made by threads of one SM is only guaranteed to be visible to threads of other SMs only across kernel calls.

Communication between host and GPU

The communication between the Host and the GPU involves the following three components: PCIe bus, Command Processor, and DMA transfers.

PCI-express bus Communication and data transfers between the system and the GPU compute device occur on the PCIex channel. Generation 1 x16 has a theoretical maximum throughput of 4 GB/s in each direction. Generation 2 x16 doubles the throughput to 8 GB/s in each direction. Transfers from the system to the GPU compute

device are done either by the command processor or by the DMA engine. The GPU compute device also can read and write system memory directly from the compute unit through kernel instructions over the PCIe bus.

The command processor The host application interacts with the OpenCL devices, through an intermediate driver layer that sends the commands to the hardware on behalf of the application. The commands to the GPU compute device are first buffered in a command queue on the host side. The command queue is then sent to the GPU compute device and processed by it. The commands are executed inorder unless explicitly specified otherwise.

DMA transfers Direct Memory Access (DMA) memory transfers can be executed separately from the command queue using the DMA engine on the GPU compute device. DMA calls are executed immediately and the order of DMA calls and command queue flushes is guaranteed. DMA transfers are executed concurrently with other GPU operations. Due to this, DMA transfers can be potentially used as a source of parallelism.

3.2 OpenCL programming model

In order to ease the effort involved in programming heterogeneous architectures, an industry consortium [20] consisting of major CPU/GPU vendors proposed OpenCL (Open Compute Language) [27] as a unified programming language. OpenCL - a dialect of C/C++, is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs and other processors. OpenCL provides a logical architecture and an API, which all compute device vendors supporting OpenCL have to implement. This frees up the programmer from knowing the details of each vendor architecture and focus on the application logic.

3.2.1 OpenCL terminologies

Work-item The smallest unit of work on an OpenCL device.

Work-group A collection of work-items, all of which are mapped to the same SM.

Wavefront A collection of work-items which execute simultaneously in a single lock-step. A wavefront is also known as a warp.

NDRange The arrangement of work-groups and work-items into a n-dimensional grid.

Kernel The C code that contains the computation logic and executed by a work-item.

Context The environment within which work-items execute. This includes devices and their memories and command queues.

Compute device The OpenCL device that executes the kernel.

Command queue A structure into which all the commands sent to a compute device are queued.

3.2.2 OpenCL memory hierarchy

OpenCL has four memory domains: private, local, global, and constant.

Private memory specific to a work-item and not visible to other work-items.

Local memory specific to a work-group and accessible only by work-items belonging to that work-group.

Global memory accessible to all work-items executing in a context, as well as to the host (read, write, and map commands).

Constant memory read-only region for host-allocated and initialized objects that are not changed during kernel execution.

Host (CPU) Memory host-accessible region for an application's data structures and program data.

An OpenCL program consists of a host part and a kernel part. The host code is run on a CPU which hosts the compute devices and performs the tasks of managing the compute devices, scheduling computations on them, transferring data to/from them, and aggregating the final computed result. The host CPU can also act as a compute device. The kernel code contains the actual computations and are run on the compute devices. The compute devices are exposed to the programmer in terms of logical threads, which are in turn grouped into thread-blocks. The computations are mapped onto these threads and thread-blocks such that each thread performs computations on a small portion of data in parallel with other threads in a SIMD manner. Each compute device have their own memory space. The data required by the computations have to be explicitly allocated on each compute device and doing this is the responsibility of the host CPU. Once computations are performed, the results have to be explicitly copied from compute device memory onto the host CPU memory. For a more detailed specification of the OpenCL standard and the API, the reader is referred to the Khronos website [27].

3.2.3 OpenCL runtime API

Device, Context, and Command queue The context is the primary management structure of OpenCL. A context encapsulates one or more devices, along with their command queues and the kernel to be run. A context can be created using `clCreateContext()`. The OpenCL devices of a particular type (CPU, GPU etc) can be discovered using the `clGetDeviceIDs()` function. A command queue can be created under a context using the `clCreateCommandQueue()`. Command queues can either be configured to process the messages inorder or out-of-order.

Kernel creation and launch OpenCL program may be created from source using `clCreateProgramWithSource()` or loaded as binaries using `clCreateProgramWithBinary()`. In either case this must be followed by `clBuildProgram()`. The OpenCL compiler may

be a dynamically linked library. A kernel can be created using `clCreateKernel()` function. Arguments can be passed to the kernel using `clSetKernelArg()`.

Memory management Global memory is allocated using clCreateBuffer(). Memory can be created with MEM_READ_ONLY, MEM_WRITE_ONLY, MEM_READ_WRITE flags. Memory is freed using clReleaseMemObject(). Variables can be allocated on the on-chip local memory by annotating the variables with the `__local__` qualifier.

Data transfers Data can be transferred from the CPU to the GPU and vice-versa using the `clEnqueueReadBuffer()` and `clEnqueueWriteBuffer()`. Another technique is to map the GPU's device buffer into CPU's address space using the `clEnqueueMapBuffer()` and once the memory is mapped, data can be read and written from the GPU using the regular `memcpy()` function. However, mapping the GPU buffer performs a copy of the data from the GPU into the CPU buffer. This is an expensive operation and should be avoided unless absolutely required. OpenCL also provides functions to transfer a non-contiguous rectangular region of memory using `clEnqueueReadBufferRect()` and `clEnqueueWriteBufferRect()` functions. This is a crucial capability we use in our work.

3.2.4 Sample OpenCL code

Figure 3.2 shows the `floyd-warshall` code as it is typically written in C for the CPU. The memory for the `path` array can be can be allocated either on the data segment through global declaration, or on the heap through dynamic allocation. The program consists of three loops - k, x and y. A quick dependence analysis can show that the k loop is serial while the y and x loop can be run in parallel.

Figure 3.3 and Figure 3.4 show the the OpenCL equivalent code for the same. The code consists of a host part and the kernel part. The execution of the program begins with the host code obtaining the OpenCL platform id and the device list. Then it goes on to create a context and a command queue for the chosen compute devices. The memory for the `path` array is explicitly allocated on the GPU's global memory. This is followed

```
1  void Floyd()
2  {
3    int x, k, y;
4
5    for(k=0; k < NUM_NODES; k++)
6    {
7      for(y=0; y < NUM_NODES; y++)
8      {
9        for(x=0; x < NUM_NODES; x++)
10       {
11         path[y][x] = ((path[y][k] + path[k][x]) < path[y][x]) ? (path[y][k] + path[k
     ][x]):path[y][x];
12       }
13     }
14   }
15 }
```

Figure 3.2: `floyd-warshall` code on cpu

by choosing of the right sizes for the NDRange.

Among the three loops of floyd-warshall, the k loop is run on the CPU sequentially, and for each iteration of this loop, the y and x loops are executed on the GPU in a DOALL parallel manner. Figure 3.4 shows the structure of the OpenCL kernel. As we can see, each array index is mapped to a thread in the OpenCL n-dimensional(2D in this case) grid. Each thread now executes one instance of the statement corresponding to its index in each dimension. At the end of execution of all serial iterations the result is copied explicitly from the GPU onto the CPU result buffer.

3.3 Polyhedral model

The polyhedral model provides a framework to compactly capture the execution sequence of statements present inside arbitrarily nested affine loops. The model provides a way to represent, analyze, and transform iterations of affine loop nests by treating them as integer points inside a convex polyhedron. For a statement inside an affine loop nest, the surrounding loop iterators along with program parameters form the dimensions of the polyhedron that represents the statement's execution domain. The upper and lower bounds of each loop iterator are represented as linear inequalities. These inequalities form

```
1  void launch_opencl_kernel()
2  {
3    clGetPlatformInfo(platform_id,CL_PLATFORM_VERSION,sizeof(str_temp),str_temp,NULL);
4    clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 2, &device_id[0], &num_devices);
5    clGetDeviceInfo(device_id[1],CL_DEVICE_NAME, sizeof(str_temp), str_temp,NULL);
6
7    clGPUContext = clCreateContext( NULL, 1, &device_id[1], NULL, NULL, &errcode);
8
9    clCommandQueue = clCreateCommandQueue(clGPUContext, device_id[1], 0, &errcode);
10
11   path_mem_obj = clCreateBuffer(clGPUContext, CL_MEM_READ_WRITE, NUM_NODES *
       NUM_NODES * sizeof(DATA_TYPE), NULL, &errcode);
12
13   localWorkSize[0] = 16; localWorkSize[1] = 16;
14   globalWorkSize[0] = NUM_NODES; globalWorkSize[1] = NUM_NODES;
15
16   for(t1 = 0; t1 < NUM_NODES; t1 ++)
17   {
18     errcode =  clSetKernelArg(clKernel, 0, sizeof(cl_mem), (void *)&path_mem_obj);
19     errcode =  clSetKernelArg(clKernel, 1, sizeof(cl_int), &t1);
20     errcode |= clSetKernelArg(clKernel, 2, sizeof(cl_int), &NUM_NODES);
21
22     errcode = clEnqueueNDRangeKernel(clCommandQueue, clKernel, 2, NULL,
       globalWorkSize, localWorkSize, 0, NULL, NULL);
23
24     clFinish(clCommandQue);
25   }
26
27   errcode = clEnqueueReadBuffer(clCommandQue, path_mem_obj, CL_TRUE, 0, NUM_NODES*
       NUM_NODES*sizeof(DATA_TYPE), &path_outputFromGpu[0][0], 0, NULL, NULL);
28 }
```

Figure 3.3: `floyd-warshall` opencl host code

```
1  __kernel void computeKernel(__global DATA_TYPE * pdm, uint numNodes, uint my_rank,
       int my_start, int my_end, int t1)
2  {
3      int xValue = get_global_id(0);
4      int yValue = get_global_id(1);
5
6      int k = t1;
7
8      DATA_TYPE oldWeight = pdm[yValue * numNodes + xValue];
9      DATA_TYPE tempWeight = (pdm[yValue * numNodes + k] + pdm[k * numNodes + xValue]);
10
11     if (tempWeight < oldWeight) {
12         pdm[yValue * numNodes + xValue] = tempWeight;
13     }
14 }
```

Figure 3.4: `floyd-warshall` opencl kernel code

```
1  for (i=0; i<N; i++)
2    for (j=0; j<N; j++)
3      A[i][j]= A[i+1][j]+A[i-1][j]; /* computation statement */
```

Figure 3.5: Sample affine loop code

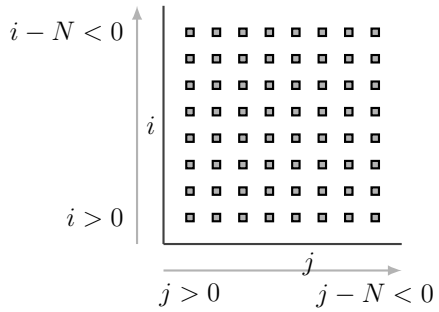


Figure 3.6: Polyhedral representation of a simple loop

the faces of the polyhedron. The set of all integer points inside a polyhedron formed by the bounds of the loop iterators surrounding a statement is called the *iteration space* of the statement. Each point in an iteration space is called an *iteration vector*, and is represented by a n-tuple, where n is the dimensionality of the iteration space. In an affine loop nest, all array accesses in the computation statements have to be affine functions of the outer loop iterators and program parameters. Such an affine function which is used as a subscript to access the elements of an array is called as an *access function*. The set of all array elements accessed by an individual access function is called the *data space* of the access function. The polyhedral framework provides libraries to perform various operations such as union, intersection, difference on these polyhedra [17, 33]. For more details on the polyhedral framework, the readers are referred to [6].

Figure 3.5 shows a two dimensional affine loop nest with a single computation statement. As we can see, the loop bounds and the array subscripts are all affine functions of the loop variables i and j. N is the program parameter whose value at runtime determines the exact number of points in the iteration space. Figure 3.6 shows the polyhedral representation of the iteration space of this statement. For ease of depiction, we have chosen N to be 8. The linear inequalities that form the bounds of the iteration space are also shown. Array subscripts i, $i + 1$, $i - 1$, and j are the access functions.

Chapter 4

BBMM - Bounding Box Based Memory Manager

This chapter forms the core part of this thesis – the Bounding Box based Memory Manager. Section 4.1 begins with the definition of a bounding box, the key idea of performing set operations on bounding boxes, the advantages using bounding boxes for GPU memory management. Section 4.2 gives an overview of each component of BBMM with an illustrative diagram. Section 4.3 describes the data allocation algorithms. Section 4.4 describes the buffer management scheme. Section 4.5 gives the details of the inter-GPU coherency scheme used in BBMM. Section 4.6 gives the overall structure of the generated code and this is followed by sections containing experimental evaluation, results and their detailed analysis.

4.1 Bounding boxes and set operations

In this section, we give the definition of bounding boxes, describe the core insights about performing set operations on them. We then justify the the advantages of using bounding boxes for GPU memory management.

Bounding box: BBMM allocates and manages data in terms *hyper-rectangles*. A hyper-rectangle is an n-dimensional counterpart of a rectangle, i.e., it has two parallel

faces for each dimension that bound the rectangle along that dimension. The smallest hyper-rectangle encapsulating a point set is called the *bounding box* of that set. In our context, a bounding box is the smallest hyper-rectangle encapsulating the elements of a multi-dimensional array which are read or written through an access function in a computation statement of a program.

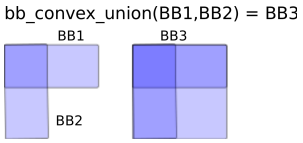
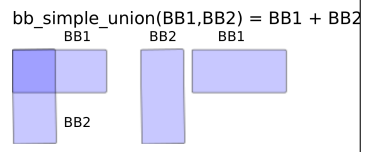
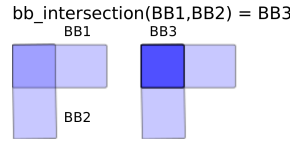
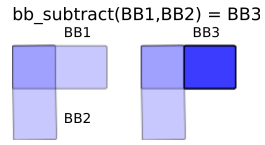
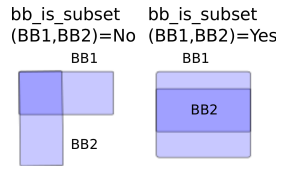
Function Name	Description	Overhead	Illustration
bb_convex_union()	Gives the convex hull of the array elements in the two bounding boxes	two bound checks in each dimension	
bb_simple_union()	Performs the union of the array elements in the two bounding boxes	simple append, dimension independent	
bb_intersection()	Performs an intersection of two bounding boxes	at most eight bound checks in each dimension	
bb_subtract()	Subtracts one bounding box from another, returning a simple union of bounding boxes	four assignments (two cuts) in each dimension	
bb_is_subset()	Checks if one bounding box is a subset of another	two bound checks in each dimension	

Table 4.1: Set operations of BBMM and their overhead

Set operation on bounding boxes: The integer points inside the bounding boxes can be subjected to common set operations like union, intersection, difference, finding subset and superset relations. Table 4.1 lists and illustrates these operations. Using them, BBMM can perform various memory optimizations such as refining compiler generated

bounding boxes, exploiting inter-tile reuse, minimizing data movement overhead etc. Note that subtraction can create multiple bounding boxes. In such cases, a simple union of these is returned. All functions of BBMM work on a simple union of bounding boxes.

The following are the advantages of using bounding boxes for GPU memory management.

1. *Negligible runtime overhead*: One key advantage of working with bounding boxes is that the set operations can be done at *runtime* merely by performing simple checks and arithmetic on their vertices. For an n-dimensional bounding box, one needs to operate on 2^n vertices. As we deal with values of n that are small enough (even for a rare case of a 5D array one needs to perform simple operations on just 32 vertices), these operations have negligible runtime overheads. This allows it to scale easily to manage a large number of GPUs.

2. *Simplicity and low cost of access functions*: An array element present in a bounding box can be accessed by simply subtracting the lower bound offsets of the bounding box from the array index in each dimension. For example, for any array `a`, an element `a[i][j]` present in a bounding box $\{lb_i,\ ub_i,\ lb_j,\ ub_j\}$ can be accessed by subtracting the lower bounds from each dimension, i.e., as `a[i-`lb_i`][j-`lb_j`]`, where lb_i and lb_j are lower bounds of the bounding box along dimension i and j respectively.

3. *Architectural support for rectangular transfers on GPUs*: GPUs (especially the ones used in high performance computing) are architecturally designed to be efficient at copying rectangular regions of memory to and from the CPU. The GPU programming models such as OpenCL and CUDA expose this capability to the user by providing rectangular copy APIs; for example, OpenCL provides `clEnqueueReadBufferRect()` and `clEnqueueWriteBufferRect()`. Using these APIs, we find that non-contiguous (strided) rectangular transfers of data between CPU and GPU, are almost as efficient as transferring contiguous bytes (Figure 4.1). This may be due to internal packing of non-contiguous elements before performing a single DMA to transfer

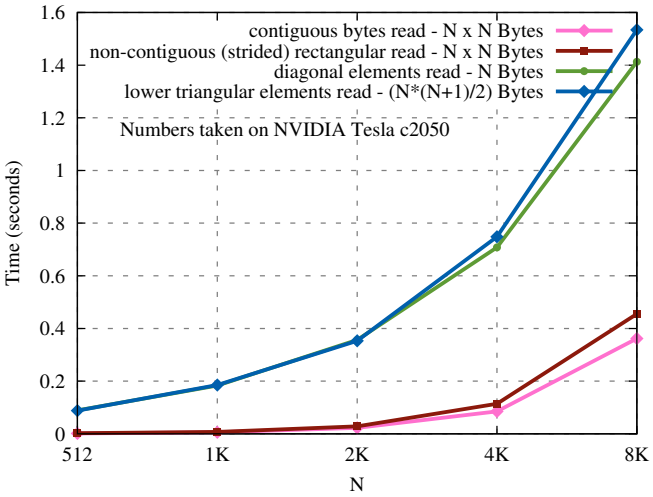


Figure 4.1: Data transfer time for various access shapes

it to destination buffers on the CPU. The same is done by the runtime driver during CPU to GPU transfers. These non-contiguous rectangular transfers are more efficient than performing multiple individual reads to transfer arbitrary non-rectangular shaped regions (say a diagonal or a triangle) of much smaller size.

4. *Non-scalability of precise data allocation techniques*: Größlinger [14] proposed a precise data allocation technique in the context of scratchpad memory management. However, the cost of access functions computed through this technique can become prohibitively high (elaborated in Chapter 6), thus making it impractical.

4.2 High-level overview of BBMM

In this section, we give a high-level view of the various components of BBMM. Figure 4.2 shows the inter-working of the input, the compile-time and the runtime components. The figure has three distinct sections separated by two dashed lines. The first section shows the input to BBMM. The second section depicts BBMM's compile time component. The third section depicts BBMM's runtime component. The arrows indicate the order of execution of each component and sub-component. Each section produces a specific output which is used as input in the next section. Below we provide additional details

for each component.

4.2.1 Input to BBMM - computation tile

BBMM expects the loop nests to be parallelized and tiled to suit GPU architectures. Hence, the input sequential C code with affine loop nests is first passed through the Pluto [31] auto-parallelizer. Pluto automatically identifies any parallelism present in the loop nests and generates a tiled and parallelized version of the loop code. In our framework, a parallelized loop nest has zero or more outer serial loops surrounding the parallel loops. The serial loops are run on the host CPU which schedules the parallel loops to run on the GPUs. The outermost parallel loop is broken down into pieces of suitable sizes (tiles) and these tiles are distributed onto available GPUs. A tile therefore is an iteration vector representing one iteration of the distributed parallel loop. All algorithms in BBMM work at the granularity of a tile.

4.2.2 Compile time component

At compile time, BBMM takes a tile as input and extracts the parameterized bounding boxes which are later refined and used at runtime. These bounding boxes are parameterized on the input tile and array. The compile-time generates the application code that refine these bounding boxes at runtime and performs various buffer management tasks. The code is generated in terms of calls to the BBMM's runtime library. The compile-time also generates the GPU kernel that accepts bounding boxes as parameters.

4.2.3 Runtime component

The runtime component of BBMM is the runtime library that is linked with the code generated at compile time. The library consists of key functions of BBMM. This includes

- Hyper-rectangular set operations used to refine the bounding boxes and thereby minimize memory allocation

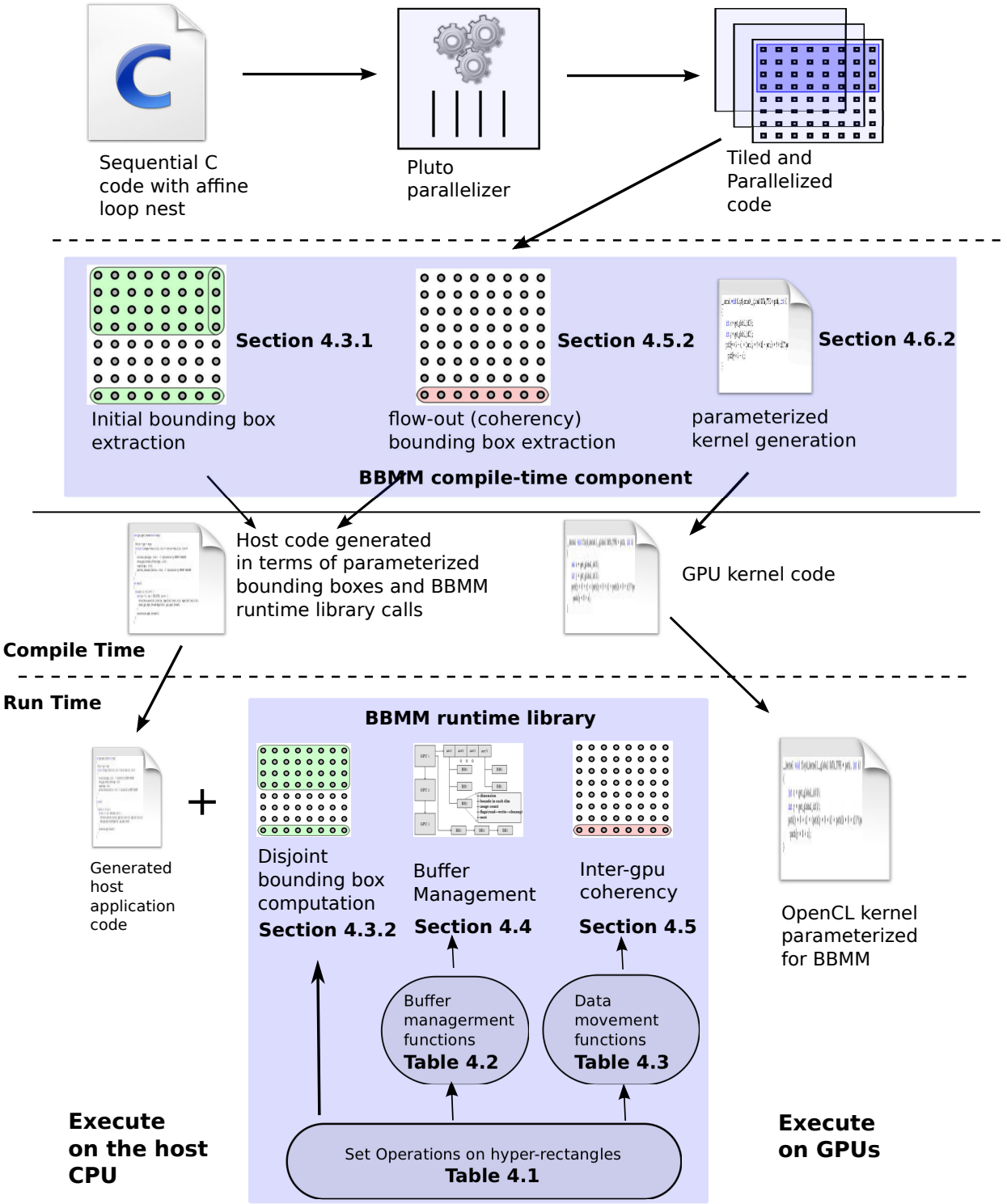


Figure 4.2: High-level overview of BBMM

- Buffer management functions that track bounding boxes on a per GPU basis and performs various memory optimizations
- Inter-gpu data movement functions that maintain coherency of data between GPUs

This runtime library is linked with the application code generated by the compile-time stage and executed on the host CPU. The parameterized kernel is executed on the GPU devices.

4.3 Data allocation scheme

In this section, we describe the data allocation scheme used in BBMM. The data allocation scheme is responsible for *identifying* the regions of array accessed by a tile and *minimizing* the memory allocated for it on the GPU. We propose a compiler-assisted runtime data allocation scheme. At compile time, an initial set of bounding boxes - one for each access function of the input tile is extracted. At runtime, these bounding boxes are further refined into a set of disjoint bounding boxes.

4.3.1 Initial bounding box extraction at compile time

This is done using a simple polyhedral technique as shown in Algorithm 1. For each read or write access function of the array, the algorithm invokes the following functions:
`get_data_polyhedron()`: Computes the data space by taking the image of the iteration space over the access function.
`get_bounding_box()`: Computes the smallest hyper-rectangle encapsulating all the points in the data space.
The set of bounding boxes returned are parameterized on the input tile and the array. The initial bounding boxes extracted at this stage could completely or partially overlap with one another. Hence, in order to avoid duplicate allocations, we need to refine these initial bounding boxes into a set of disjoint bounding boxes. We note that this refinement operation can be performed at compile time using the polyhedral library. However doing

so at compile time might result in over-allocations since the library has to allocate for the maximum value of the bounds to ensure correctness. Also, the library might choose to perform a large number of splits depending on the type of initial bounding boxes. At runtime, we have the precise bounds for the initial bounding boxes and as mentioned before, the rectangular set operations can be performed with negligible overhead. Hence, we choose to perform the bounding box refinement at runtime.

Algorithm 1: `extract_initial_bounding_boxes()`

Input: Computation tile $\vec{t}$, Array a
$S_a^{init} = \phi$
for each *read or write access function* f_a^i **do**
  dp_a^i = get_data_polyhedron(f_a^i)
  bb_a^i = get_bounding_box(dp_a^i)
  add bb_a^i to S_a^{init}
Output: S_a^{init}, the set of initial bounding boxes

Algorithm 2: `get_disjoint_bounding_boxes()`

Input: S_a^{init} - Set of initial bounding boxes for tile $\vec{t}$ and array a
$S_a^{disjoint} = \phi$
for each *bounding box* bb_a^{init} *in* S_a^{init} **do**
  $bb_a^{rem} = bb_a^{init}$
  for each *bounding box* bb_a^{disj} *in* $S_a^{disjoint}$ **do**
    $bb_a^{intersect}$ = bb_intersection(bb_a^{rem}, bb_a^{disj})
    bb_a^{rem} = bb_subtract(bb_a^{rem}, $bb_a^{intersect}$)
  add bb_a^{rem} to $S_a^{disjoint}$
Output: $S_a^{disjoint}$, the set of disjoint bounding boxes for array a

4.3.2 Disjoint set of bounding boxes at runtime

The key steps of the data allocation scheme are performed at runtime as shown in Algorithm 2. The input is the set of initial bounding boxes now substituted with the *actual* values for tile and array parameters (since we have this information at runtime). On each of these exact bounding boxes, the algorithm uses the set operations described in Table 4.1 to subtract out the portions which are already present in the set of disjoint

bounding boxes. This is done as shown in lines 2 to 6. The portion that still remains is added to the disjoint bounding box set.

4.3.3 Example

Figure 2.3 illustrates the data allocation scheme for `floyd-warshall`. Figure 2.3a shows the iteration space of a single tile. For illustration purpose, we have chosen $N = 8$ and $k = 7$. Figure 2.3c shows the initial bounding boxes; one for each distinct access function. `path[i][j]` covers an area equal to the size of the tile. `path[i][k]` covers $N - 1^{th}$ column. `path[k][j]` covers the $N - 1^{th}$ row. Figure 4.3 shows the initial bounding boxes as it looks in the generated code. The generated bounding boxes are parameterized on the input tile represented by the iteration vector (t0, t1, t2, t3) and the input array. The set of initial bounding boxes is input into Algorithm 2. Figure 2.3e shows the result of running Algorithm 2 on initial bounding box list. For the chosen tile, the bounding box of `path[i][k]` is a subset of the bounding box of `path[i][j]` whereas bounding box of `path[k][j]` is disjoint from both. Hence the algorithm returns two disjoint bounding boxes BB0 and BB1 as shown.

4.3.4 Discussion

Access function split and warp divergence: In some cases a single access function of an array can get split among multiple bounding boxes due to the disjoint operation. In these cases, a runtime check has to be made in the computation kernel on the GPU to determine the bounding box which contains a particular array index. If different threads of a warp have to access different bounding boxes, it will result in warp divergence causing loss of parallelism. Though this seems like a problem in theory, in practice this is not an issue due to the following reasons:

- For programs with uniform dependences like stencil computations, the adjacent threads (forming the warp) access adjacent memory locations. Hence, they almost always access data from the same bounding box, i.e., they take the same control

```
1  list extract_initial_bounding_boxes_for_path_in_kernel_0(Device dev, int t0, int t1,
       int t2, int t3, void * array)
2  {
3      // the iteration vector (t0,t1,t2,t3) represents the tile. t1 corresponds to the
4      // serial iteration k. t3 represents the tiled parallel dimension.
5      ....
6      // path[i][j]
7      struct bounding_box * bb0 = bb_alloc(2);
8      bb0->array = array;
9      bb0->flags = BBMM_FLAGS_READ_WRITE;
10     bb0->bp[0].lb = (TILE_SIZE * t3) + (0);
11     bb0->bp[0].ub = (TILE_SIZE * t3) + ((TILE_SIZE-1));
12     bb0->bp[1].lb = 0;
13     bb0->bp[1].ub = +1*N-1;
14     add_to_list(&init_bb_list, bb0);
15     // path[i][k]
16     struct bounding_box * bb1 = bb_alloc(2);
17     bb1->flags = BBMM_FLAGS_READ;
18     bb1->array = array;
19     bb1->bp[0].lb = (TILE_SIZE * t3) + (0);
20     bb1->bp[0].ub = (TILE_SIZE * t3) + ((TILE_SIZE-1));
21     bb1->bp[1].lb = (1 * t1) + (0);
22     bb1->bp[1].ub = (1 * t1) + (0);
23     add_to_list(&init_bb_list, bb1);
24     // path[k][j]
25     struct bounding_box * bb2 = bb_alloc(2);
26     bb2->array = array;
27     bb2->flags = BBMM_FLAGS_READ | BBMM_FLAGS_CLEANUP;
28     bb2->bp[0].lb = (1 * t1) + (0);
29     bb2->bp[0].ub = (1 * t1) + (0);
30     bb2->bp[1].lb = 0;
31     bb2->bp[1].ub = +1*N-1;
32     add_to_list(&init_bb_list, bb2);
33
34     return init_bb_list;
35 }
```

Figure 4.3: Compiler generated function that returns initial bounding boxes for a tile

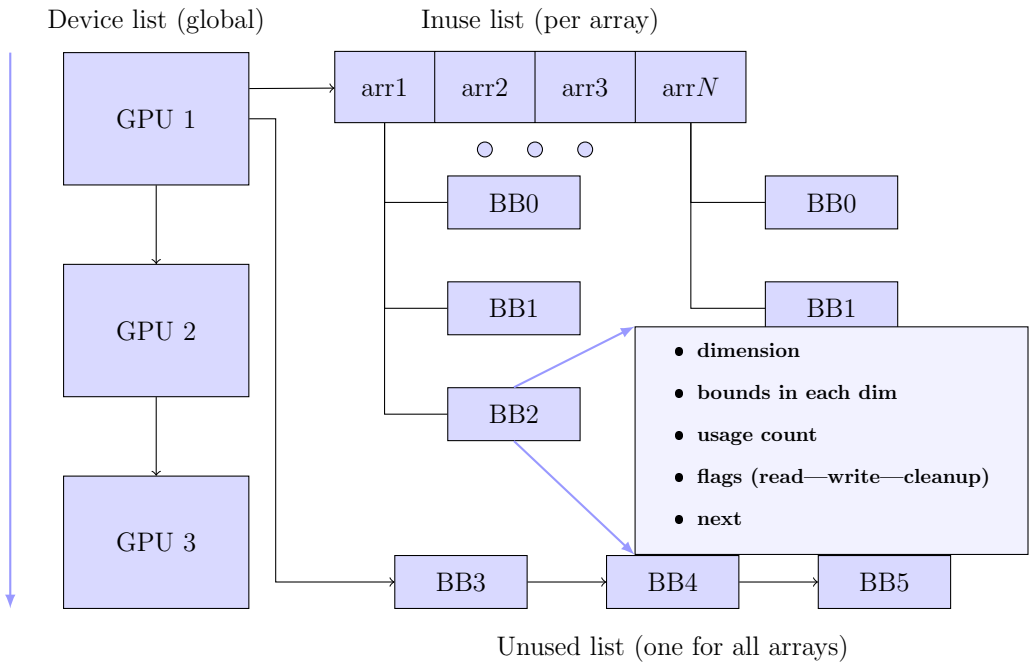


Figure 4.4: Buffer management component of BBMM

flow path, resulting in no performance loss.

- In order to incur a performance loss due to warp divergence, an access function should both be non-uniform (so that adjacent threads forming a warp access different memory regions) and split among multiple bounding boxes. For this loss to be significant, a large number of access functions should have been split among multiple bounding boxes. This happens very rarely in practice. Even when it does happen, the performance loss will not be prohibitively significant.

In Section 4.9, we provide results that support the above reasoning.

4.4 Buffer management

In this section, we describe the techniques used in the buffer management component of BBMM. This component is responsible for tracking the bounding boxes allocated on each GPU, reduce duplicate allocations, maximize inter-tile data reuse and provide capability to work with data sizes larger than the available GPU memory.

4.4.1 Design overview

Figure 4.4 shows the design of the buffer management component. For each GPU, the buffer manager maintains two lists of bounding boxes: (1) inuse list (2) and unused list. The ones that are currently being read or written by a tile will be in inuse list. The unused list is used to free up memory on a GPU whenever required and is maintained in the least recently used (LRU) order. Each bounding box is associated with a usage count, indicating the number of compute tiles currently using it. A bounding box will be in unused list only if its usage count is zero. The usage count will be important in a setting where multiple parallel tiles are running simultaneously on the same GPU and these tiles are sharing the same bounding box. In such cases, the bounding box should not be freed (say, to make space) until all the tiles are done using the bounding box. It also has flags to indicate whether it will be only read, written, and what all needs to be cleaned up. The memory manager provides various inexpensive runtime functions on the bounding boxes. Table 4.2 lists the important ones.

Function name	Description
bb_alloc()	Allocates a bounding box on a device, making space if needed
bb_present()	Checks if a given bounding box is already present on a device
bb_readin()	Initializes data into a bounding box, with intra-device transfers if possible
bb_cleanup()	De-allocate bounding boxes that are no longer required

Table 4.2: Functions provided by the buffer manager

4.4.2 Inter-tile data reuse

Using hyper-rectangles allows BBMM to find the subset and superset relation between bounding boxes at runtime, with negligible overhead. It uses this functionality, to reuse the data already present on a device and thereby minimize data transfers from CPU to GPU. Data reuse happens at two levels. In the first level, BBMM checks if a bounding box is already present or fully subsumed on the GPU before it allocates a new bounding box. This check is performed by the `bb_present()` function. This avoids redundant allocations.

Algorithm 3: `bb_readin()`

Input: Device dev, Array a, Bounding box bb_a^{new}

1 **for each** *bounding box* bb_a^{dev} *on the device* **do**
2 $bb_a^{intersect}$ = bb_intersection(bb_a^{new}, bb_a^{dev})
3 **if** $bb_a^{intersect}$ *is not empty* **then**
  /* part of the data required for this bounding box is already present on the device */
4 intra_device_copy(bb_a^{new}, bb_a^{dev}, $bb_a^{intersect}$)
5 bb_a^{new} = bb_subtract(bb_a^{new}, $bb_a^{intersect}$)
6
7 **if** bb_a^{new} *is not null* **then**
  /* some portion of the bounding box is not yet read in. copy this data from cpu */
8 cpu_to_gpu_copy(bb_a^{new})

The second level of reuse happens for partially subsumed bounding boxes. When a new bounding box is allocated on a device, it has to be initialized with the latest data. Instead of doing this entirely with the data from the CPU, BBMM reuses data that is already present on a device through intra-device copies from one bounding box to another. `bb_readin()` (Algorithm 3) is used to perform this. The function uses the `bb_intersection()` to find the intersection of the new bounding box with the bounding boxes already present on the GPU. The intersecting area is copied to the new bounding box through an intra-device copy and the intersecting area is subtracted out from it. Major GPU programming frameworks such as OpenCL and CUDA provide functions to perform intra-device copies. For example, OpenCL provides `clEnqueueCopyBuffer()` function that can perform this copy. The process is repeated for all bounding boxes already present on the device. The leftover portion is initialized with the data from the CPU.

Data reuse as a cost metric: In a dynamic computation placement scheme, the device on which a tile is run is determined dynamically using a cost heuristic which chooses the device that can minimize the overall costs of execution. Intra-device data reuse can be used as one such metric. When there is an option to run a tile on any one of multiple GPUs, running it on the GPU which maximizes intra-device data reuse would reduce

the data movement overhead. BBMM enables such a scheme by providing functions to compute the reuse factor for a given tile and a device using the technique described above. However in this thesis we do not present results with such a dynamic scheme.

Cleaning up unwanted bounding boxes: If an initial bounding box is only parameterized on outer serial loop iterations, then it can be freed immediately after those iterations complete. During the initial bounding box extraction, the compiler checks whether this is the case and sets the cleanup flag in the bounding box. Such bounding boxes are freed by the memory manager at the end of the same outer serial loop iteration in which it was allocated. The `bb_cleanup()` function performs this task.

4.4.3 Freeing up space on a GPU – box-in and box-out

The function `bb_alloc()` can free space on the GPU for allocating new bounding boxes, if required. It does this by freeing up bounding boxes from the unused list in the LRU order. LRU policy was chosen as it captures the temporal locality property of a program, which will allow BBMM to work well with a compiler transformation that optimizes for temporal reuse. If the bounding box about to be freed was earlier written on the GPU, its data is copied onto the CPU before it is freed. When a future tile requires this data, it is copied from the CPU back to the GPU. This in essence is like the page-in and page-out process of the CPU, except that the granularity of memory is in terms of bounding boxes. Hence, we call this *box-in* and *box-out*. This feature allows applications to automatically work with data size much larger than the *combined* memory sizes of all the GPUs. However, box-out and box-in operations are highly expensive since it involves significant data transfers. Hence, for this feature to be of use, the tiles for which we are freeing space need to have sufficient computations in them to compensate for the box-in and box-out overhead *over and above* the time it takes for the tile to run on the CPU. If this is not the case, it will be more efficient to run the tiles on the CPU itself. In our experiments, embarrassingly parallel applications like `blackscholes` achieved excellent scaling due to the fact that they had a very high compute-to-copy ratio. However, applications which had an outer serial loop performed worse because a single parallel

tile within each outer serial iteration did not have sufficient compute-to-copy ratio. The results and detailed analysis for this feature are presented in Section 4.9.

4.4.4 Relationship between tiles, bounding boxes and multiple GPUs

Tiles of a parallel loop can execute simultaneously on either the same GPU or different GPUs. Data allocation function is run for each tile and bounding boxes are identified at the granularity of an individual tile. In the case when multiple tiles are running on the same GPU, BBMM performs an optimization to ensure that there is no redundant allocation for fully-overlapping bounding boxes. If bounding boxes of different tiles only partially overlap, then no such optimization is performed on them, and two copies of the same array element can be present in two different bounding box on the same GPU. In such a case the inter-GPU coherency scheme of BBMM ensures that multiple copies the array elements within the same GPU is also in sync. This is possible because, in BBMM, the coherency function is called at the granularity of a tile i.e, BBMM performs *inter-tile* coherency.

4.5 Inter-GPU coherency

GPUs in a multi-GPU machine do not share address space. Hence, if tiles distributed across different GPUs access same elements of an array, a copy of the element will be present on multiple GPUs. Such a copy can also exist within the same GPU if two tiles executing on the same GPU require the same array element and the bounding boxes of those two tiles containing the array element do not fully overlap. Dependence analysis during parallelism extraction stage ensures that in a parallel phase of execution i.e, within the same outer serial loop execution, tiles do not have any data dependences. However, across serial loop iterations, flow dependences (RAW dependences) can exist. In such a case, a write to an array element by a tile will cause the other copies (if they exist) of that element to become stale. Hence, explicit data transfers have to be performed

to keep all the copies of the array element in sync before the next serial loop iteration begins.

4.5.1 High-level overview of BBMM's coherency scheme

BBMM uses a compiler assisted runtime coherency scheme. The compiler uses dependence analysis to identify precise coherency data for each tile. This data is parametric on the input tile iterators. This information is passed onto the runtime as part of the compiler generated code. The runtime utilizes this information to obtain the exact coherency data by substituting the parametric tile iterators with the exact values since it now has that information. The runtime now orchestrates the inter-GPU data movement as follows. It first copies out the coherency data from the source GPU onto the CPU's copy of that array. It then checks if any other GPU has a bounding box containing that element. If it finds such a bounding box, it immediately updates it with the copied out value. This ensures that whenever a bounding box is present on a GPU, its *necessary elements* (the elements read by at least one future tile) are always kept updated with the latest data. If no such bounding box is found (which can happen if no tile reading that data has yet been run) the updated values are retained on the CPU.

4.5.2 Details of BBMM's coherency scheme

The data transfer to and from the GPU is expensive since it is via the PCIex bus which has a limited bandwidth (8 GB/s). This data transfer needs to be efficient in order to ensure that the overhead of coherency does not override the benefits of distributing the computation onto multiple GPUs. Efficient data movement techniques for distributed memory setups is an orthogonal problem to ours and many current and past works have tried to address it [1, 2, 7, 9, 10, 12, 22]. In this thesis, we use the state-of-the-art data movement scheme for distributed-memory scenarios proposed by Dathathri et al [12]. This scheme called Flow-Out Partitioning (FOP) first identifies the data to be transferred from a source tile called the *flow-out* set. The flow-out set is further refined

using a technique called `source-distinct partitioning` in which data transfers due to multiple dependences are grouped together such that all the elements from a partition are required by all the receivers of that partition. This eliminates both unnecessary and duplicate data transfers inherent in other data movement schemes. The scheme has been demonstrated to work efficiently with both distributed memory clusters and heterogeneous systems. The complete details of this scheme are beyond the scope of this thesis and the reader is referred to [12] for the same. We describe the adaptation of this scheme and how we extend the idea of bounding boxes for inter-tile data transfers.

Rectangular transfers and flow-out bounding boxes

Since rectangular transfers (both contiguous and non-contiguous) are efficiently supported on the GPUs 4.1, we approximate the flow-out sets into flow-out bounding boxes. We then copy a flow-out bounding box onto the CPU with a single rectangular read. On the CPU, the precise flow-out elements are unpacked from the rectangular buffer onto the CPU's copy of the array. The fact that BBMM natively works with rectangles and its ability to identify subsumed bounding boxes enables it to utilize the rectangular data transfer efficiently thereby minimizing its overhead.

Generation of flow-out bounding boxes at compile time

The compile-time component of BBMM extracts the flow-out sets as flow-out bounding boxes. These bounding boxes are parameterized on the input tile and array. A function similar to the one shown in Figure 4.3 is generated.

Data movement orchestration at runtime

The BBMM runtime library contains the data movement orchestration component. At runtime, three distinct data transfers are performed. Table 4.3 lists the functions that perform these data transfers.

gpu_to_cpu_flowout(): This function copies the partitioned flow-out sets out of the bounding box of the source tile onto a copy of the array maintained by the host CPU.

Function Name	Description
gpu_to_cpu_flowout()	The data items that are written by a tile and read by other tiles due to RAW (flow) dependences are copied out from the source GPU into the CPU.
cpu_to_gpu_flowin()	The updated data is copied from the CPU onto the bounding boxes of the target tiles
gpu_to_cpu_writeout()	The data items that are written last by this tile and will be part of the final output are copied from the source GPU onto the CPU

Table 4.3: Functions provided by the data movement component

The CPU always has the latest copy of the flow-out sets. This ensures that a future tile depending on this flow-out set can be provided the required data from the CPU itself. When the exact flow-out sets are non-rectangular this function copies the entire bounding box onto the CPU and then unpacks the precise elements on the CPU's copy of the array.

cpu_to_gpu_flowin(): The flow-out data copied out to the CPU, now needs to be copied into any overlapping bounding boxes on other GPUs. To do this, we perform an intersection of the flow-out bounding boxes with the bounding boxes allocated on other GPUs. For each intersecting portion, a CPU to GPU data transfer is performed, thereby updating the destination bounding boxes to the latest state. In cases where the exact flow-in sets are non-rectangular, correctness is ensured by first copying the flow-in set into a temporary staging buffer on the GPU, and using a flow-in bitmask which indicates the elements within the staging buffer that has to be copied onto the destination bounding boxes.

4.6 Host and kernel code generation

In this section, we describe how all the components of BBMM come together as part of the generated host and kernel code.

4.6.1 Structure of the generated host code

The overall structure of the host code that drives the GPU kernel execution on a multi-GPU machine is shown in Algorithm 4. The execution begins from the outer serial loops. For each iteration of this loop, the set of parallel tiles are distributed among the available GPUs. This can be a one-dimensional or multi-dimensional block or block-cyclic distribution. The code following the distribution of tiles, is executed in the context of a worker thread which manages a particular GPU. For each tile distributed to that GPU the worker thread runs the code shown between lines 3 and 18. For each array accessed in the tile, the set of bounding boxes to be allocated is obtained using Algorithm 2. Each bounding box is checked for presence on the GPU using `bb_present()`. If it is not found, the bounding box is allocated on the GPU with `bb_alloc()` and initialized with `bb_readin()`. Once the bounding boxes are ready, their usage counts are incremented so that the box-in box-out logic does not remove them to free space if need be. The tile is then scheduled for computation on the GPU. Once the computations complete, the inter-tile data transfers are performed using the functions described in Table 4.3. Following this, the usage count of the bounding boxes are decremented and any bounding box which has a usage count of zero is moved to the unused list to be either reused or freed up for next iteration. At the end of the outer serial loop iteration, the bounding boxes

that are marked for cleanup using `bb_cleanup()`.

Algorithm 4: `Structure of the generated host code`

```
1  for each iteration of the outer serial loop i_s do
2      distribute the parallel tiles of i_s among the GPUs
       /* below code is executed in the context of a host worker thread that manages the GPU */
3      for each parallel tile t⃗ of i_s allocated to GPU dev do
4          S = φ
5          for each array a accessed in t⃗ do
6              S_a = get_disjoint_bounding_boxes(t⃗,a)
7              for each bounding box bb in S_a do
8                  if !bb_present(dev, a, bb) then
9                      bb_alloc(dev, a, bb)
10                     bb_readin(dev, a, bb)
11                 increment_usage_count(bb)
12             S = S ∪ S_a
13         compute(t⃗, dev, S)
14         gpu_to_cpu_flowout(t⃗, S)
15         cpu_to_gpu_flowin(t⃗, S)
16         gpu_to_cpu_writeout(t⃗, S)
17         for each bounding box bb in S do
18             decrement_usage_count(bb)
19     bb_cleanup(dev, i_s)
```

4.6.2 Structure of the parameterized GPU kernel

GPU kernel generation is an orthogonal problem to ours and works such as [5, 29, 41] have focused on automatically generating optimized GPU kernels from sequential CPU codes. In BBMM, kernel generation is not a core problem we address. Rather, we just parameterize the GPU kernels to accept bounding boxes. The access functions are outlined so that they access data from the input bounding boxes. Figure 4.5 shows the general structure of the parameterized GPU kernel. At compile time, each kernel is generated with a list of bounding boxes as parameters. At runtime, these parameters

```
1  void ComputeKernel0(int split0, DATA_TYPE * buf0, int buf0_lb0, int buf0_ub0, int
       buf0_lb1, int buf0_ub1, int split1, DATA_TYPE * buf1, int buf1_lb0, int buf1_ub0,
        int buf1_lb1, int buf1_ub1, ....)
2  {
3      DATA_TYPE * var_wacc_0 = KERNEL0_var_WACC(split0, buf0, buf0_lb0, buf0_ub0,
       buf0_lb1, buf0_ub1, idx0, idx1);
4      DATA_TYPE var_racc_0  = KERNEL0_var_RACC(split1, buf1, buf1_lb0, buf1_ub0,
       buf1_lb1, buf1_ub1, idx0, idx1);
5      ...
6      // do the computation using values obtained above.
7      *var_wacc_0 = var_racc_0 + ...
8  }
```

Figure 4.5: General structure of the parameterized GPU kernel

are set to appropriate disjoint bounding box buffers using `clSetKernelArgs()` function. Each access function in the kernel is passed with information of whether its bounding box is split. Each access function in the kernel is outlined with a wrapping macro that checks if the bounding box corresponding to it is split. If it is not split (i.e., splitX = 0, which is the majority case) the macro dereferences the buffer pointer associated with the bounding box using the indices that are subtracted by offsets of that bounding box. If the access function is split, then the macro checks each bounding box bounds to determine which bounding box the index being accessed belongs to; that buffer is dereferenced with the appropriate indices.

4.7 Implementation

BBMM's implementation has a compiler component and a runtime component. The compile-time component is integrated with the polyhedral source to source compiler Pluto [8]. The input to Pluto is a serial C code containing arbitrarily nested affine loop nests. Pluto creates a polyhedral representation of the program and identifies the serial and parallel loops in it. It also extracts array access and dependence information from the input code. The parallelized code is tiled with user provided tile sizes. Choosing appropriate tiles sizes automatically is an orthogonal problem. Doing this, in part, requires parametric tiling, which is an ongoing research in the polyhedral area [39].

Hence, BBMM currently requires the user to provide the tile sizes at compile time and to ensure that the size of data required by a single tile is less than the global memory size of the GPU on which the tile executes. BBMM's compile-time component uses this tiled and parallelized code as input and generates the following code. The functions to return the set of initial bounding boxes and flow-out bounding boxes as shown in Figure 4.3, a function to implement the data allocation scheme shown in Algorithm 4, and the parameterized OpenCL kernel. The runtime component of BBMM is implemented as a standalone C library that can be linked with any C/C++ application. The library implements the algorithms and functions described in earlier sections. The library is generic and does not contain any references to platform specific code such as CUDA or OpenCL. Hence, it can be used with either of them. For the results in this thesis, we made the following choices of tile distribution and scheduling:

- a static one-dimensional block distribution of parallel tiles among GPUs is used,
- at any time, only one tile is active on a given GPU, and multiple parallel tiles distributed to the same GPU are executed sequentially in lexicographical order,
- tiles across different (independent) parallel loop nests are executed sequentially in program order across different kernel calls.

The reason for placing these restrictions is to focus our experimentation on the efficiency and effectiveness of the core parameters of BBMM such as allocation sizes, reuse exploitation, inter-gpu coherency, box-in/box-out, and runtime overheads of the library and generated kernel. However, we note that BBMM's techniques can be made to work with any choice of tile distribution and scheduling schemes, whether static or dynamic.

4.8 Experimental setup and benchmarks

The experiments were run on an Intel Xeon multicore server consisting of 12 Xeon E5645 cores (2-way SMP of hex-core) running at 2.4 GHz. The server includes 3 NVIDIA Tesla C2050 and 1 NVIDIA Tesla K20 graphics processors connected on the PCI-express bus.

Program	Source	Dep pattern	A	B	Data size on 1 GPU (1X)		C	D	E
					Array sizes	Size (GB)			
floyd	Polybench	non-uniform	1	2	16384 x 16384	2.0	2	yes	0.05%
heat2d	Pochoir	uniform	2	2	12288 x 12288	2.25	4	yes	0.10%
fdtd2d	Polybench	uniform	3	2	10240 x 10240	2.4	2	yes	0.06%
heat3d	Pochoir	uniform	2	3	512x512x512	2.0	4	yes	0.04%
lu	Polybench	non-uniform	1	2	16384 x 16384	2.0	3	yes	0.07%
adi	Polybench	uniform	3	2	8192 x 8192	1.5	2	yes	0.01%
mvt	Polybench	EP	3	2	20480 x 10240	1.5	1	no	0.01%
bscholes	NVIDIA	EP	3	1	67,108,864	1.5	1	no	0.01%

Table 4.4: Programs used for evaluation A: number of arrays. B: maximum dimensionality of arrays C: maximum number of bounding boxes for any array D: subsumed bounding boxes present? E: BBMM runtime overhead as a percentage of overall execution time

The Tesla C2050 have 2.5 GB of global memory each and the K20 has 5 GB of global memory. For our experiments we chose to limit the memory usage of K20 to 2.5 GB to maintain uniformity. We thus have a combined GPU memory size of 10 GB. NVIDIA driver version 304.64 supporting OpenCL 1.1 was used as the OpenCL runtime.

The programs for evaluating BBMM was drawn from a variety of benchmarks such as Polybench [32], Pochoir [38], and the NVIDIA GPU SDK [25]. The main criteria for selection of test programs was for them to have affine loop bounds and affine access functions. To test all the features of BBMM, we chose benchmarks that have different dependence patterns (uniform, non-uniform, embarrassingly parallel (EP)), and different array dimensionalities. The selected programs are listed in Table 4.4.

4.9 Evaluation parameters and results

In this section, we describe the parameters on which we evaluate BBMM, the insights we hope to gain from each of them, and the experimental results and analysis for the same.

4.9.1 Overhead of the runtime library

This is an important parameter that measures the execution overhead of BBMM runtime library functions, giving an insight into the cost of the various bounding box set

operations and the memory management functions listed in Tables 4.1 and 4.2. To compute this, the time taken for memory management is first obtained by measuring the time spent inside the library, throughout a program's execution. The overhead is then computed as a percentage of the overall execution time of the program. Specifically:

$total\_time = memory\_mgmt\_time + compute\_time + flowout\_time + flowin\_time + writeout\_time$

$overhead\_percentage$ = ($memory\_mgmt\_time$ / $total\_time$) * 100

Table 4.4 (column F) shows the overhead percentage. For all programs, this does not exceed 0.1% of the total execution time and is thus insignificant.

4.9.2 Performance of programs with data scaling

Data scaling refers to the scenario when dataset sizes of a program are increased at the same rate as the combined memory size of the processing elements (GPUs in this case), i.e., the data size per GPU remains constant. This parameter is similar to weak scaling, but the emphasis is on memory utilization rather than on workload. Hence, for this parameter, we report the speedup per outer sequential loop iteration. The per-iteration speedup of a program is computed by dividing the per-iteration execution time with 2X, 3X and 4X data sizes with the per-iteration execution time with 1X data size. The value of X for each program is specified in Table 4.4. This cancels out the effect of the algorithmic complexity of the program on speedup calculations. The per-iteration execution time considered for this calculation includes all overheads, i.e., data allocation time, computation time on the GPUs, flow-out time, flow-in time and write-out time. BBMM's schemes directly affect all, except the pure computation time on the GPUs.

Figure 4.6 shows the data scaling results on 2, 3 and 4 GPUs. For every additional GPU added to the experiment, the data size is increased proportionally so that the data size per GPU is constant. For all programs except `adi`, we see that the speedup is around 1 with a geometric mean speedup of 0.94, indicating near ideal data scaling. However, for `adi`, we see a significant slowdown with the increase in the number of GPUs. This is because, `adi` has an enormous amount of inter-device data movement at the end of each

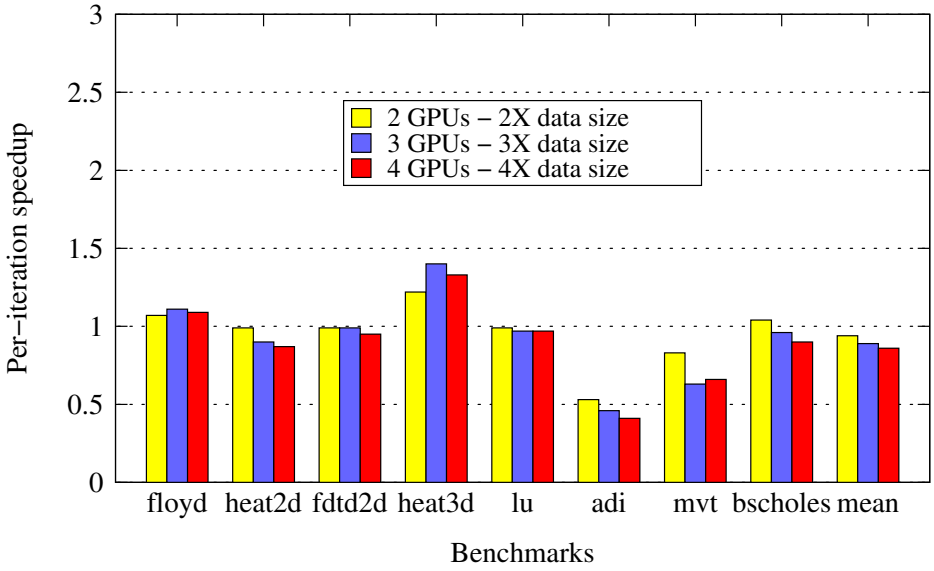


Figure 4.6: Performance with data-scaling

serial iteration to maintain coherency among the GPUs. This causes the compute-to-copy ratio to be very small for this program, resulting in poor scaling. This high volume of data movement is due to the dependence patterns in the program and independent of our memory management schemes.

4.9.3 Comparison of data allocation sizes

We compare the data allocation sizes of BBMM with that of a convex bounding box approach and the theoretical exact sizes required. The convex bounding box for a tile is obtained by performing the convex union of the initial bounding boxes. The exact sizes required are manually calculated. The sizes are reported as a percentage of the array size. Comparing with the convex approach gives an insight into the maximum reduction in data allocation sizes due to the use of disjoint operation. Comparing with the exact sizes, tells us how good BBMM's allocation sizes are compared to any possible manual effort.

Figure 4.7 shows the reduction in data allocation sizes as result of using disjoint bounding boxes. For `floyd` and `lu`, which have non-uniform data access pattern, using disjoint bounding boxes greatly reduces the allocation sizes – up to 75% on our 4 GPU

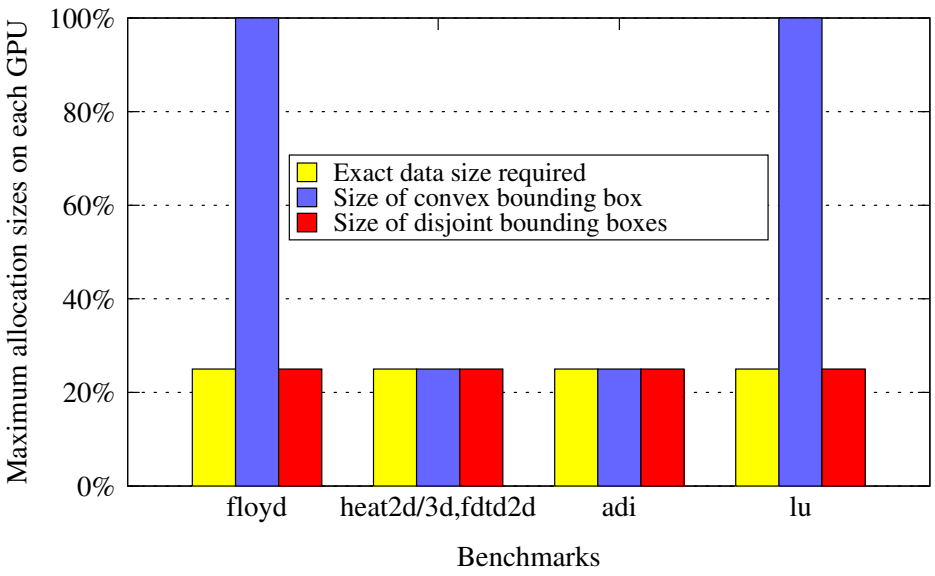


Figure 4.7: Allocation size comparison between exact, convex and disjoint bounding box schemes

machine as compared to convex bounding boxes. However, for the class of programs such as stencils and linesweep (`heat2d`, `heat3d`, `fdtd2d`, `adi`) that have uniform accesses, the difference in sizes are negligible. Also, for all the programs, the allocation sizes due to disjoint bounding boxes equals the exact required data sizes (computed manually). This shows that BBMM's automatically generated data sizes for these programs are as good as any possible manual programming effort.

4.9.4 Benefits of inter-tile data reuse

This gives an insight into the extent of the benefit gained by exploiting data reuse across different tiles of a program. We report the speedup of programs when this optimization is enabled, over the same programs with this optimization disabled.

Figure 4.8 shows the speedup obtained with inter-tile data reuse enabled. The baseline is the execution time of the programs without enabling this optimization. Exploiting inter-tile data reuse yielded a mean speedup of 5.4 over without-reuse case. `heat2d`, `heat3d`, `lu`, and `fdtd2d` have tiles whose bounding boxes are already fully subsumed

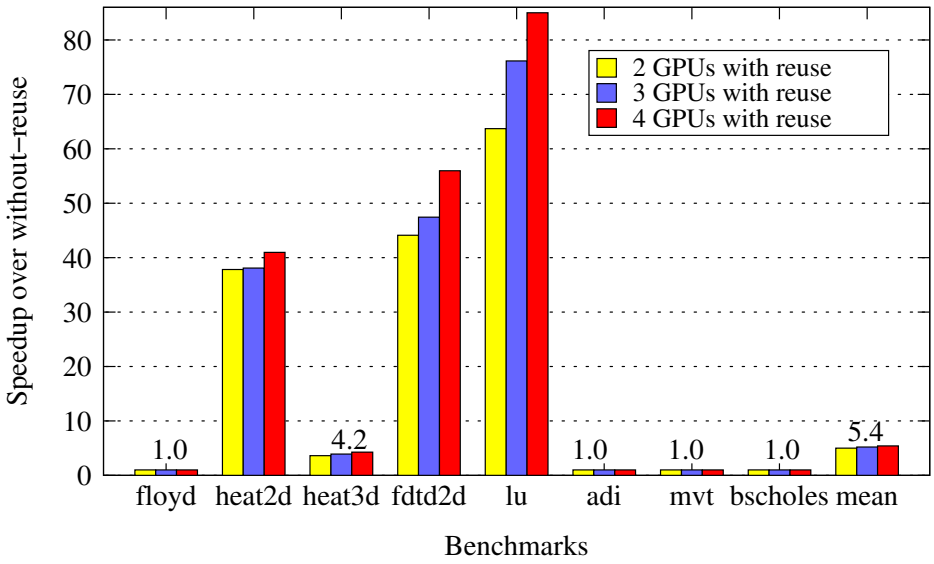


Figure 4.8: Speedup with inter-tile reuse as compared to without-reuse

inside bounding boxes of other tiles. When inter-tile data reuse is disabled, these subsumed bounding boxes are also allocated separately. The flow-out sets have to be now copied to these as well which results in significant amount of data movement overhead. `blackscholes` and `adi` do not have subsumed bounding boxes and `floyd` has very small flow-out sets. Hence, they are not affected by this optimization.

4.9.5 Effect of access function split

This measures the effect of access function split (hence, possible warp divergence) on the execution times of the programs. Among our test programs, the stencils (`heat2d`, `heat3d`, `fdtd2d`) and linesweep(`adi`) which have uniform dependences undergo a split for some of their access functions. For programs with non-uniform accesses such as `floyd` access functions do not split. Hence, in order to investigate the effect of warp divergence, we modified our scheme as follows:

- For stencils/linesweep, we forced our algorithm to return `convex` bounding boxes so that all the access functions now access from a single convex bounding box and hence they do not split.

- For `floyd`, we forced BBMM to always treat *all* its access functions as split (which

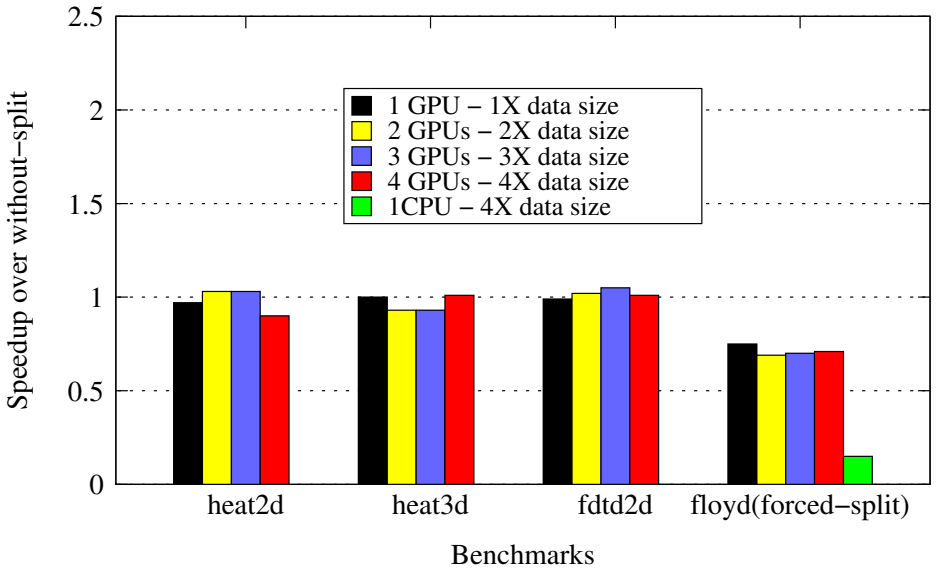


Figure 4.9: Performance with access function splits as compared to without-split

is the worst possible case that can happen). Hence in this case, there is a conditional check for every array access. This gives an insight into the worst case performance of the generated kernel.

Figure 4.9 shows the performance when access functions are split across multiple bounding boxes. The baseline is the execution time of the same program without any access function split. As explained in Section 4.3.4, we see that stencils do not have any performance degradation even when the bounding boxes are split. However, for `floyd` we see a 40% degradation in performance. Even though this is a significant degradation, we have to note that this is for the (forced) worst case scenario of every access function getting split, which rarely happens in practice. Also, even with a 40% percent degradation, we note that the performance on a GPU is much better than a tiled and parallelized version of `floyd` running on the 12-core system.

4.9.6 Benefit of box-in and box-out

As explained in section 4.4.3, for this feature to be beneficial, one has to ensure that the tiles to be run on GPUs have sufficient computation to compensate for the data movement overheads. If not, there could be performance degradation. Hence, we selected `matmul`

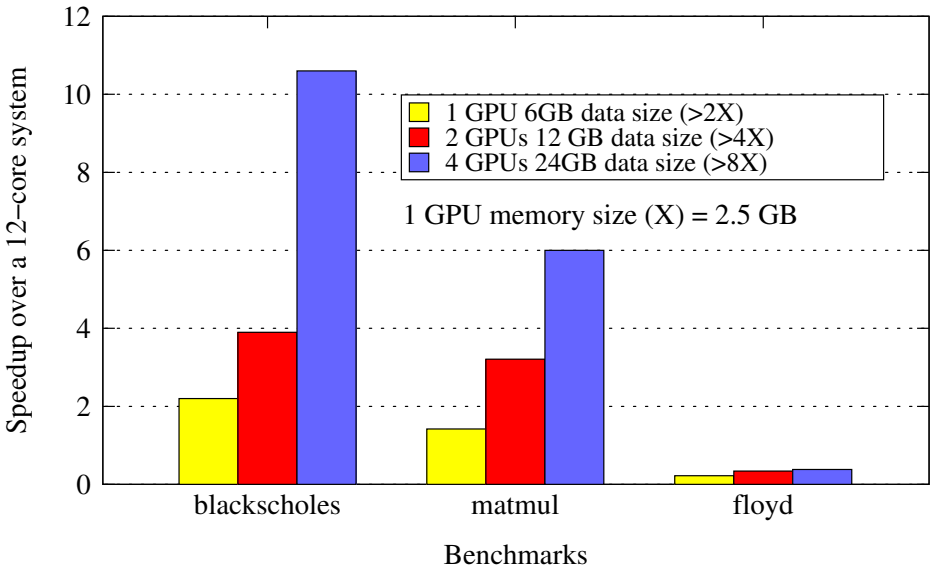


Figure 4.10: Speedup with box-in and box-out over a 12-core system

and `blackscholes` that have this property (high compute-to-copy ratio) to demonstrate the benefits of box-in/box-out and `floyd` to demonstrate the performance degradation. Also, we compare the GPU performance with that of a multi-core CPU, since we are interested in understanding the maximum data sizes up to which using GPUs (rather than the CPU itself) would be beneficial.

Figure 4.10 shows the speedup of programs with box-in and box-out over running on 12-core system. The CPU version of the code was automatically tiled and OpenMP-parallelized using Pluto. For `blackscholes` and `matmul` which are embarrassingly parallel, we noticed significant speedup compared to their CPU versions. However, for programs like `floyd` which have outer serial loop, there was a slowdown in the GPU performance. One way to increase the compute-to-copy ratio for `floyd`, is to tile the outer serial loop (or the time dimension in case of stencils). In this thesis, we do not present results with such a tiling.

4.9.7 Comparison with manual code

We provide a comparison of BBMM's automatically generated memory management code with manually written OpenCL [27], OpenACC [26] codes. This gives insights into

the efficiency of the code generated by BBMM in terms of allocation sizes and coherency costs, as compared to fully hand-optimized codes. We also evaluate BBMM's box-in/box-out feature when compared with manually written code picked from StarPU [3] suite on a single GPU. This can yield useful insights about the maximum data sizes a program can work with.

Comparison with manually written OpenCL codes

For this comparison, we chose `floyd` as a representative example of non-uniform access pattern, `heat2d` for its uniform access pattern and inter-tile data reuse potential, and `blackscholes` for its embarrassingly parallel structure. `floyd` and `heat2d` require coherency at the end of each outer serial loop iteration. These three programs together cover all the characteristics present in the other programs of our test suite. The codes were written with the following optimizations:

- The computations were manually distributed equally among the GPUs.
- The data allocation sizes for each tile of computation was the theoretical minimum.
- The volume of data moved for coherency was theoretical minimum.
- Reuse exploitation was theoretical maximum i.e., no data which was already on the GPU was re-allocated or re-initialized.

Figure 4.11 shows the relative execution time of BBMM as compared to manually written OpenCL code on 2 and 4 GPUs. In each case, the code generated by BBMM performed almost as efficiently (minimum being 88%) as the manually optimized code. In each case, the size of data allocated by BBMM is equal to the size of manually allocated data. In other words, the allocation sizes are exact. In case of `floyd` and `heat2d` the volume of data moved by BBMM due to coherency is also equal to that of manually written code. The slight slowdown in BBMM's code can be attributed to two factors: (1) overhead of BBMM's memory management data structures, (2) additional memory accesses present in the generated kernel. However, considering the significant

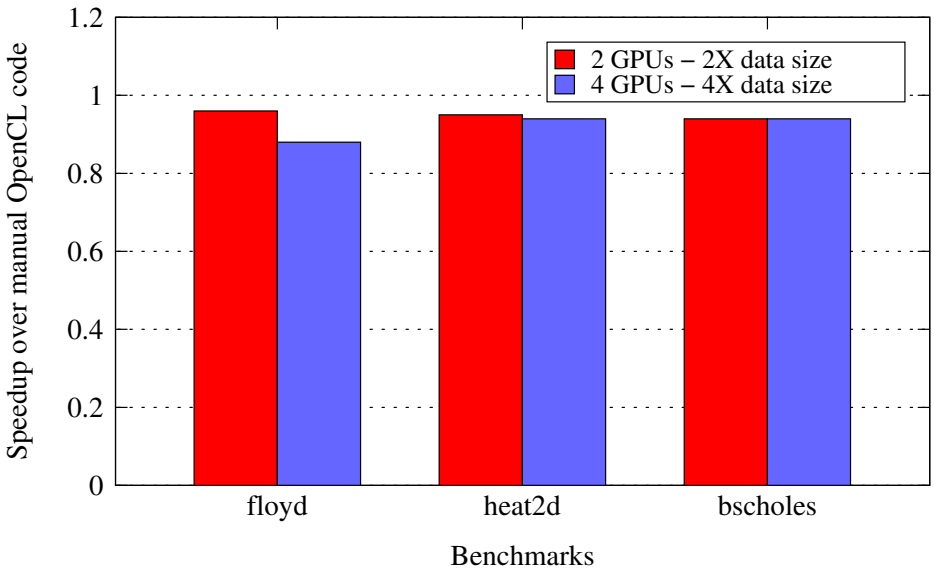


Figure 4.11: Performance normalized to manually written multi-GPU OpenCL code

programming efforts involved in writing the manual code, this slight slowdown can be considered acceptable.

Comparison with manually written OpenACC codes

OpenACC [26] does not have support for *automatic* array distribution and coherency. Hence, in our manual OpenACC codes, we performed the array distribution manually and allocated the needed sub-arrays on the GPUs using `cudaMalloc()`. We then used the `deviceptr` clause to pass these data pointers to the OpenACC kernel. The coherency management was done manually using `cudaMemcpy()`. PGI [30] compiler version 12.10, supporting OpenACC 1.0 specification, was used to compile the codes. As in the case of OpenCL, the allocation sizes and coherency volume was kept to theoretical minimum. Again, we chose `floyd`, `heat2d` and `blackscholes` as representative programs for non-uniform, uniform and embarrassingly parallel applications.

Figure 4.12 shows the comparison of the execution times of BBMM and OpenACC on 2 and 4 GPUs. For `floyd` and `heat2d`, BBMM has a speedup of 1.6× and 1.3× respectively, over manual OpenACC code in spite of having the same volume of data allocation and coherency. This can be attributed to the OpenACC kernel execution

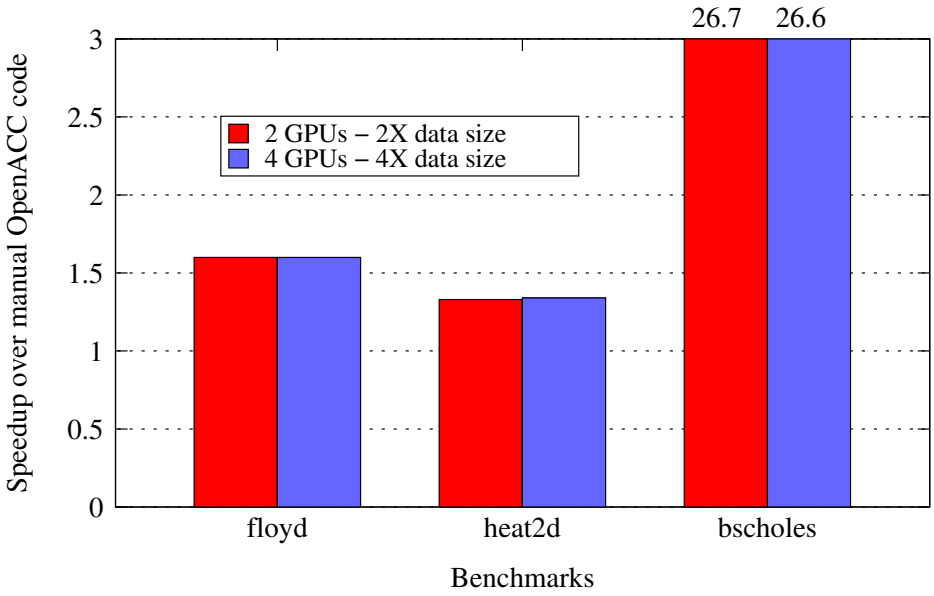


Figure 4.12: Performance normalized to manually written multi-GPU OpenACC code

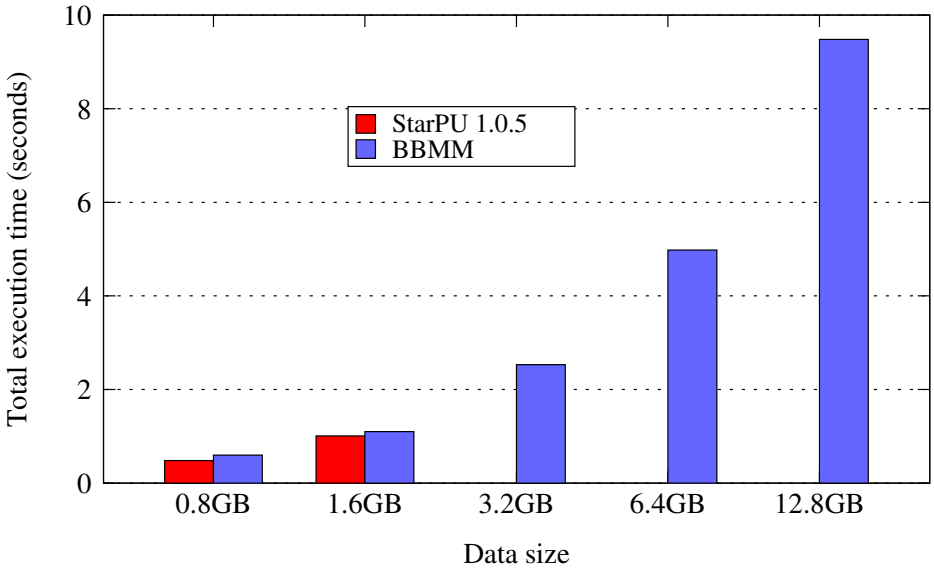


Figure 4.13: Comparison with StarPU for `mvt` on 1 GPU

overheads. This overhead is even more significant in `blackscholes` where BBMM has a speedup of over 26× over OpenACC. In this case, OpenACC selected inefficient grid and block sizes. We suspect that the significant slowdown in its performance is due to this. In spite of our best efforts to manually correct this with `num_gangs` and `num_workers` clauses, the OpenACC compiler ignored our directives and continued to use its internally computed values.

Comparison with manually written StarPU code

Figure 4.13 provides a comparison of automatic data allocation with BBMM with a manually written code for `mvt` taken from the StarPU framework. The manual version of the code was taken as-is from the StarPU 1.0.5 example suite. We see that, until the data size was within the GPU memory size, the performance of BBMM's automatically generated code was on par with the manually version of the code. Beyond that data size, StarPU could not allocate data whereas with BBMM, box-out and box-in kicked in causing data to be automatically freed up on the GPU to make space for new bounding boxes. This resulted in an ideal scaling up to the tested data size of 12.8 GB ($> 5\times$ the single GPU memory).

Chapter 5

Data movement scheme: Details and further optimizations

In this chapter, we provide more details of the efficient inter-GPU data movement techniques used in BBMM. We first provide an overview of the techniques themselves and then provide the detailed experimental evaluation and results. We then describe a novel technique that we have developed to maximize compute-copy overlap which further reduces the data movement overhead, and enables applications to achieve ideal strong scaling.

5.1 Brief description of the schemes

In this section, we describe the working of two state-of-the-art automatic data movement schemes for distributed-memory systems which form the basis for the scheme used in BBMM – the Flow-Out scheme (FO) and further improvement on it called the Flow-Out Partitioning scheme (FOP).

5.1.1 The Flow-Out (FO) scheme

The `flow-out` scheme was proposed by Bondhugula [7] as an efficient and automatic technique for generating data-movement code for distributed memory scenarios. In this

scheme, the data that needs to be moved out of a source tile due to RAW dependences - called the `flow-out set`, is computed using advanced polyhedral techniques. The flow-out set is parameterized on the source tile and is computed for a given array and for all dependences arising due to the read and write accesses to that array. The scheme also automatically identifies the list of receiver tiles to whom the flow-out set needs to be sent. At runtime, the scheme can work with any tile distribution function that maps the tiles onto the actual compute devices. The drawback of this scheme arises from the fact that, the entire flow-out set is sent to all receiver devices, even though each receiver does not need to receive all the elements in the flow-out set.

5.1.2 The Flow-Out Partitioning (FOP) scheme

The drawback of FO scheme was overcome by our work in Dathathri et al [12]. This scheme called Flow-Out Partitioning (FOP), partitions the flow-out set such that all the elements from a partition are required by all the receivers of that partition. The partitioning of the flow-out set is achieved by using a technique called *source-distinct* partitioning of the RAW dependences, which is described below.

source-distinct partitioning: A source-distinct partitioning of dependences partitions the dependences such that the region of data flowing due to all the dependences in a partition is the same and the region of data flowing through two different partitions are disjoint.

In order to partition dependences, it is necessary to determine whether the regions of data that flow due to two dependences overlap, i.e., whether the region of data written by the source iterations of one dependence overlaps with that of the other. This can be determined by an explicit dependence test between the source iterations of one dependence and the source iterations of another dependence. Such a dependence might not be semantically valid (e.g., when there is overlap in the regions of data that flow due to dependences with the same source statement). It is just a virtual dependence between two dependences, that captures the overlap in the regions of data that flow due to those dependences.

If a virtual dependence exists between two dependences, the iterations of the original dependence that share the source iterations with the virtual dependences are subtracted out from the original dependences. The virtual dependences, along with the remainder of the original dependences are now source-distinct. The process is repeated until no new partitions can be formed. The data flowing due to each of the source-distinct partitions are now sent to receivers that require at least one receiver in the partition. Since the flow-out partitions are disjoint and each of them combines the data to be communicated due to multiple dependences, this scheme reduces duplication.

5.2 Experimental Evaluation

We evaluated FO and FOP schemes independently on two major GPU vendor platforms - NVIDIA and AMD. Below we present the setup, benchmarks and results for both of these.

5.2.1 Experimental setup

Intel-NVIDIA system: The Intel-NVIDIA system consists of an Intel Xeon multicore server consisting of 12 Xeon E5645 cores running at 2.4 GHz. The server has 4 NVIDIA Tesla C2050 graphics processors connected on the PCI express bus, each having 2.5 GB of global memory. NVIDIA driver version 304.64 supporting OpenCL 1.1 was used as the OpenCL runtime. Double-precision floating-point operations were used in all benchmarks. The host codes were compiled with gcc version 4.4 with -O3.

AMD system: The AMD system consists of a AMD A8-3850 Fusion APU, consisting of 4 CPU cores running at 2.9 GHz and an integrated GPU based on the AMD Radeon HD 6550D architecture. The system has two ATI FirePro V4800 discrete graphics processors connected on the PCI express bus, each having 512 MB of global memory. Since these GPUs do not support double-precision floating-point operations, we use single-precision floating-point operations in all benchmarks. The data sizes are chosen such that the entire array data fits within each GPU's global memory. AMD driver version

9.82 supporting OpenCL 1.2 was used as the OpenCL runtime. The host codes were compiled with g++ version 4.6.1 with -O3.

5.2.2 Benchmarks

We evaluate FO and FOP for `floyd`, `lu`, `fdtd-2d`, `heat-2d` and `heat-3d` benchmarks. These benchmarks are taken from the same source as listed in Table 4.4. Since the focus is on data movement, we have not considered the programs that are embarrassingly parallel.

5.2.3 Evaluation

We consider the following combination of compute devices: (i) 1 CPU, (ii) 1 GPU, (iii) 2 GPUs, (iv) 4 GPUs. We evaluate FO and FOP on the Intel-NVIDIA system for all these cases. On the AMD system, we evaluate FO and FOP for (i), (ii) and (iv) cases, using only the discrete GPUs. In the first two cases, the devices run the entire OpenCL kernel. For cases (iii) and (iv), kernel execution is partitioned across devices and equally distributed (block-wise).

5.2.4 Results

Table 5.1 shows results obtained on the Intel-NVIDIA system. For all benchmarks, the running time on 1 GPU is much lower than that on the 12-core CPU. This running time is further improved by distributing the computation onto 2 and 4 GPUs. For all benchmarks, we see that FOP significantly reduces communication volume over FO. The computation tile sizes directly affects the communication volume (e.g., 32× for `floyd`). For the transformations and placement chosen for these benchmarks, we manually verified that FOP achieved the minimum communication volume. This reduction in communication volume results in a corresponding reduction in execution time facilitating strong scaling of these benchmarks, as shown in Figure 5.1 – this was not possible with the existing FO. For example, FO for `heat-3d` has very high communication overhead and

does not scale beyond two GPUs. For `floyd` and `lu`, FO scales up to 2 GPUs, but not beyond it. However, FOP easily scales up to 4 GPUs for all benchmarks. The extent of scalability for `lu` is not as much as the other programs. This is due to the fact that `lu` has a low compute-to-copy ratio. In other words the cost of inter-GPU data movement significantly offsets the gains of computation partitioning. This is because, `lu` has non-uniform dependences, due to which its flowout data has to be copied out to each of the other GPUs in the system, leading to significant data movement overhead. This property of `lu` is more visible in the FO scheme where the cost of datamovement completely overrides the cost of computation partitioning resulting in significant slowdown rather than speedup.

Table 5.2 shows results obtained on the AMD system. The OpenCL functions used to transfer rectangular regions of memory are crucial for copying non-contiguous (strided) data efficiently. We found these functions to have a prohibitively high overhead on this system. This compelled us to use only those functions which could copy contiguous regions of memory. Hence, we present results only for `floyd`, `heat-2d` and `fdtd-2d` since the data to be moved for these benchmarks is contiguous. For all benchmarks, the running time on 1 GPU is much lower than that on the 4-core CPU. FO does not perform well on 2 GPUs for `heat-2d` and `fdtd-2d` since these benchmarks have a low compute-to-copy ratio and the high volume of communication in FO leads to a slowdown.

The FOP scheme, on the other hand, performs very well on 2 GPUs, yielding a near-ideal speedup of 1.8× over 1 GPU for all benchmarks.

5.3 Further optimizations: Maximizing compute-copy overlap

In this section, we describe further optimization to the data movement technique described in previous section and demonstrate its benefits.

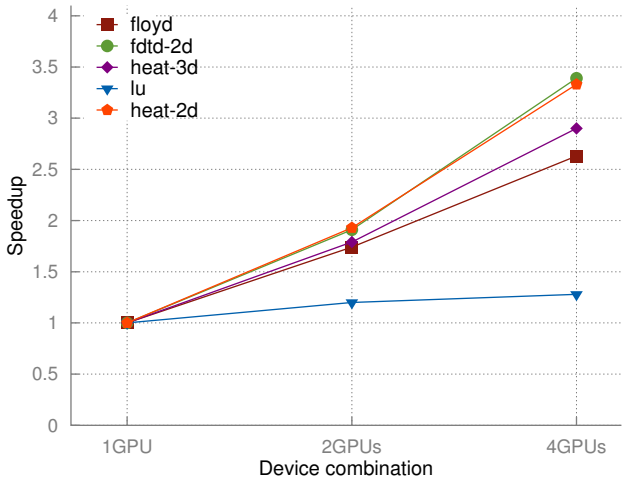


Figure 5.1: FOP – strong scaling on the Intel-NVIDIA system

Bench mark	Problem sizes	Tile sizes	Dev Comb	Total execution time				Total communication volume(in GB)		
				-	FOP	FO	Speed up	FOP	FO	Reduction factor
floyd	10240 x 10240	32 x 32	1 CPU	890s	–	–	–	–	–	–
			1 GPU	113s	–	–	–	–	–	–
			2 GPUs	–	65s	104s	1.60	1.6	51.0	32
			4 GPUs	–	43s	107s	2.49	3.1	102.0	32
lu	11264 x 11264	256 x 256	1 CPU	412s	–	–	–	–	–	–
			1 GPU	77s	–	–	–	–	–	–
			2 GPUs	–	64s	147s	2.30	0.7	62.0	83
			4 GPUs	–	60s	208s	3.47	1.2	63.0	51
fdtd-2d	4096 x 10240 x 10240	32 x 32	1 CPU	1915s	–	–	–	–	–	–
			1 GPU	397s	–	–	–	–	–	–
			2 GPUs	–	207s	236s	1.14	0.9	22.0	22
			4 GPUs	–	117s	164s	1.40	2.2	62.0	28
heat-2d	4096 x 10240 x 10240	32 x 32	1 CPU	1112s	–	–	–	–	–	–
			1 GPU	266s	–	–	–	–	–	–
			2 GPUs	–	138s	157s	1.14	0.6	21.0	32
			4 GPUs	–	80s	124s	1.55	1.9	62.0	32
heat-3d	4096 x 512 x 512 x 512	32 x 32 x 32	1 CPU	3080s	–	–	–	–	–	–
			1 GPU	1932s	–	–	–	–	–	–
			2 GPUs	–	1086s	1379s	1.26	16.0	512.0	32
			4 GPUs	–	670s	1658s	2.47	49.0	1535.4	32

Table 5.1: Results on the Intel-NVIDIA system

Bench mark	Problem sizes	Tile sizes	Dev Comb	Total execution time				Total communication volume(in GB)		
				-	FOP	FO	Speed up	FOP	FO	Reduction factor
floyd	10240 x 10240	32 x 32	1 CPU 1 GPU 2 GPUs	1084s 512s –	– – 286s	– – 305s	– – 1.07	– – 0.8	– – 25.0	– – 32
fdtd-2d	4096 x 5120 x 5120	32 x 32	1 CPU 1 GPU 2 GPUs	1529s 241s –	– – 133s	– – 242s	– – 1.82	– – 0.2	– – 2.15	– – 17
heat-2d	4096 x 8192 x 8192	32 x 32	1 CPU 1 GPU 2 GPUs	3654s 256s –	– – 142s	– – 353s	– – 2.49	– – 0.25	– – 8.0	– – 32

Table 5.2: Results on the AMD system

5.3.1 Compute-copy overlap

GPUs used in HPC have a DMA engine that can perform data transfers independent of the computations going on in the GPU. This allows the programmer to overlap data movement associated with a program with a non-conflicting computation. Overlapping data movement with kernel execution helps in hiding the data movement overhead. Many works in the literature have recognized this fact [21, 28, 35, 37] and utilize this technique for performance gains. The most commonly used technique is called *double-buffering*. In this technique, a program uses two sets of input and output buffers – one set for the currently executing kernel and another set for the kernel to be executed next. While the current kernel is executing, the host component of the program commands the DMA engine to copy the input data for the next kernel into its input buffer. Simultaneously, it commands that the output of the previous kernel be transferred out into the CPU buffer. The PCIex bus can be simultaneously used in both directions. Once the current kernel completes, the buffers are switched i.e, the buffer set of the completed kernel become the buffer set of the next kernel. Two sets of buffers are used to ensure that the data updated by the simultaneous computation and copy do not conflict.

5.3.2 Compute-copy overlap in our framework

In our framework, each tile executes on a GPU through a kernel call. Within a single outer serial loop iteration, multiple parallel loop tiles can be distributed to the same GPU. Since the tiles can be run in parallel, they do not have any conflicting data accesses (dependences) among them. This provides two key benefits:

- The parallel tiles can be executed in any order.
- The flow-out data of a tile can be copied safely while another parallel tile is executing on the same GPU.

Hence, we do not need to use the double buffering technique.

5.3.3 Implementing compute-copy overlap

We create two different command queues for each GPU; compute queue and data queue. The kernel execution commands are enqueued in the compute queue and the data read and write commands are enqueued in data queue. The runtime can process the different command queues concurrently. For each tile we enqueue a kernel launch command on the compute queue. We then enqueue a non-blocking read command for the flowout bounding boxes on the data queue. This command is given an event dependency on the previous kernel launch. This ensures that the flowout does not start until the associated kernel computation is completed. Meanwhile, the host thread can go about enqueuing subsequent computation and read commands. Once all the compute and copy commands are enqueued, the GPU management thread on the host waits for the event status of a flowout command to reach completion. Then the cpu_to_gpu_flowin() function is invoked which copies the flowout bounding boxes onto the destination GPUs. We again note that this function can be invoked while a kernel execution is going on in the destination GPU. This is because, since each GPU is executing a distributed parallel loop, the data read by one GPU with in an outer serial iteration will not conflict with the data read or written by another GPU in the same iteration. Figure 5.2a depicts the comparison

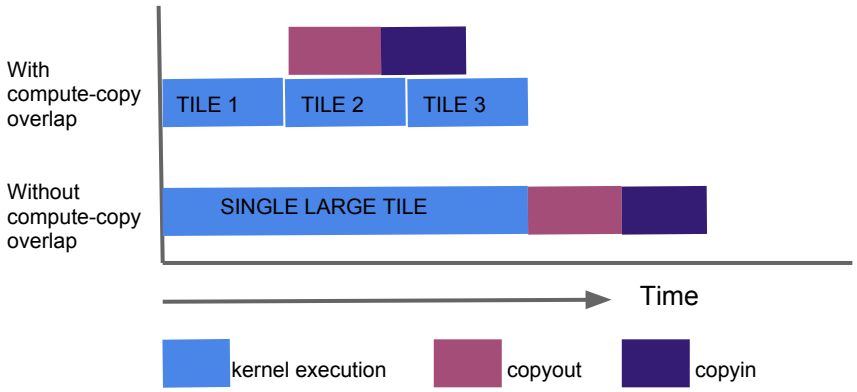


Figure 5.2: Benefit of Compute-copy overlap

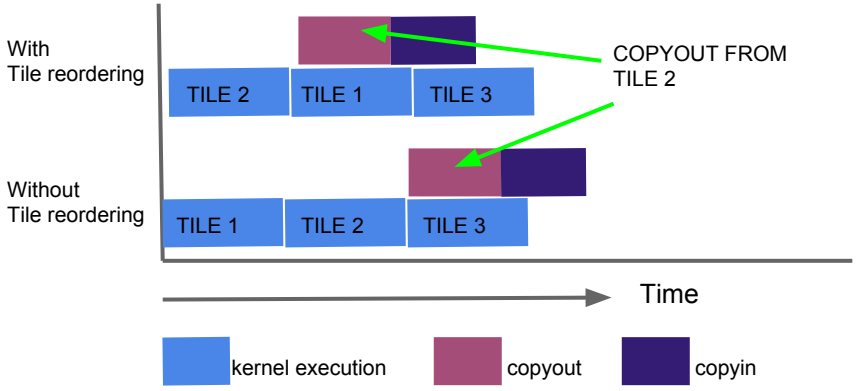


Figure 5.3: Compute-copy overlap with and without tile reordering

of data movement overheads with and without compute-copy overlap. Without overlap, the flowout starts only after the entire computation completes. With overlap however, it starts immediately after the execution of the tile which has flowout data.

5.3.4 Maximizing compute-copy overlap

Even though compute-copy overlap provides significant improvements to the overall execution time of programs, the extent of the improvement with just the earlier described technique is not always maximum i.e., the copy time is not overlapped to the maximum possible extent with respect to the computation time of the remaining tiles. In order to gain maximum overlap, the copyout process should start as early as possible with respect to the computation process. We notice that, in most programs each computation tile does not have the same amount of coherency data to be moved into other GPUs. Some

tiles do not have any data to be copied out at all. In such cases, scheduling the tiles in the decreasing order of their flowout data sizes can help achieve maximum compute-copy overlap. Hence, at runtime, we compute the size of the flowout bounding boxes of each tile and reorder the tile execution (within the same iteration) to be in the decreasing order this size. Figure 5.3a depicts the execution time with and without tile reordering. As we can see, in cases when tile2 or tile3 has flowout data, the copyout time can extend beyond the actual computation time even with overlap. However, with tile reordering we can ensure that the flowout can start at the earliest possible time point and hence can have maximum overlap with computation time. Our experiments with this technique on the Intel-NVIDIA setup showed significant performance improvements for all the benchmarks. We present these results in the next section.

Prog	Problem sizes	Tile sizes	Device comb	Exec time						
				-	FOP	FO	FO-cc	FO-cctr	FOP-cc	FOP-cctr
floyd	10240 x 10240	32 x 32	1 CPU	890s	–	–	–	–	–	–
			1 GPU	113s	–	–	–	–	–	–
			2 GPUs	–	65s	104s	78s	59s	61s	59s
			4 GPUs	–	43s	107s	98s	87s	40s	37s
lu	11264 x 11264	256 x 256	1 CPU	412s	–	–	–	–	–	–
			1 GPU	77s	–	–	–	–	–	–
			2 GPUs	–	64s	147s	111s	111s	45s	45s
			4 GPUs	–	60s	208s	201s	201s	55s	55s
fdtd-2d	4096 x 10240 x 10240	32 x 32	1 CPU	1915s	–	–	–	–	–	–
			1 GPU	397s	–	–	–	–	–	–
			2 GPUs	–	207s	236s	216s	199s	202s	199s
			4 GPUs	–	117s	164s	140s	117s	110s	100s
heat-2d	4096 x 10240 x 10240	32 x 32	1 CPU	1112s	–	–	–	–	–	–
			1 GPU	266s	–	–	–	–	–	–
			2 GPUs	–	138s	157s	150s	134s	137s	133s
			4 GPUs	–	80s	124s	100s	93s	76s	67s
heat-3d	4096 x 512 x 512 x 512	32 x 32 x 32	1 CPU	3080s	–	–	–	–	–	–
			1 GPU	1892s	–	–	–	–	–	–
			2 GPUs	–	1043s	1350s	1344s	1168s	1039s	997s
			4 GPUs	–	603s	1294s	1260s	1236s	593s	548s

Table 5.3: Results of compute-copy overlap on Intel-NVIDIA System (cc: compute-copy overlap, cctr: compute-copy overlap with tile reordering)

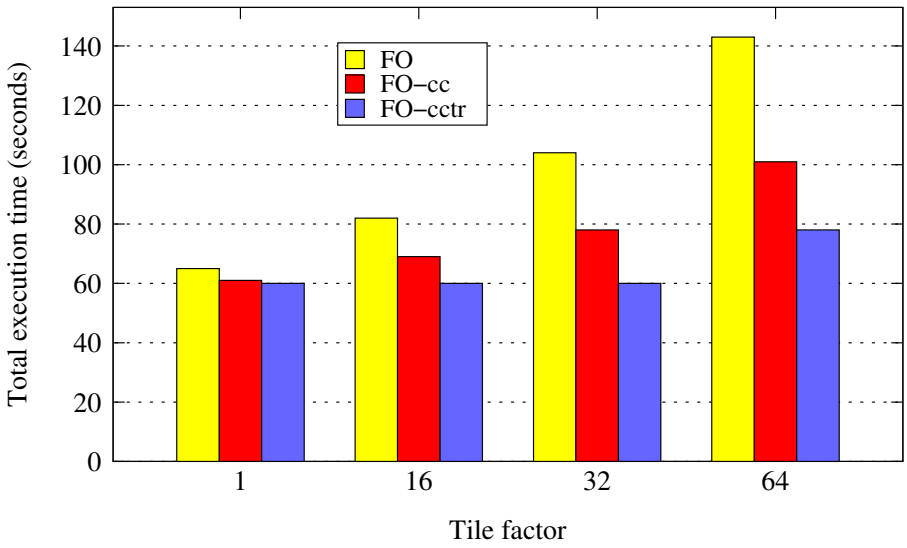


Figure 5.4: Performance of tile reordering with varying tile sizes for `floyd-warshall` with FO scheme on a 2-GPU setup

5.4 Experimental results

Table 5.3 shows the performance of the benchmark programs with compute-copy overlap – with and without tile reordering, for both FO and FOP schemes. We have shown the numbers for FO scheme as well to get an insight on the benefits that our overlap technique can provide when the volume of data transferred is high. We see that, for all programs compute-copy overlap numbers are significantly better than those without it. This is shown in FO-cc and FOP-cc columns (*cc* stands for compute-copy overlap). We see that FO which had a high overhead of data movement to begin with benefits greatly from our technique. This shows that, programs with high volume of data movement will benefit greatly from this technique. These numbers are further improved by reordering the tiles in the decreasing order of their flowout bounding box sizes. This is shown in the FO-cctr and FOP-cctr columns (*cctr* stands for compute-copy overlap with tile reordering) respectively.

For `floyd`, only the tile accessing the k^{th} row will have flowout data. In the FO case, the flowout data will be as large as the size of the tile. This adds significant overhead to the overall execution time. Hence, FO case gains a significant speedup of 1.33× with compute-copy overlap over the without-overlap case. In the FOP case, `floyd`

requires only the k^{th} row to be copied out. In a lexicographical scheduling of tiles, the tile producing this flowout data starts from the first tile and moves towards later tiles as k increases. For higher values of k, this causes flowout time to exceed beyond the computation time of all tiles. Hence, a tile schedule in which the tile accessing the k^{th} row is always executed first will cause the flowout overhead to hidden to the maximum extent possible. For `heat2d`, `heat3d`, and `fdtd2d`, in the FOP case, only the first and last tiles distributed to each GPU within each serial loop iteration have flowout data. With tile reordering, these two tiles get scheduled first thereby maximizing the overlap of the flowout transfer time with the computation time of the remaining tiles. For our chosen tiles sizes these three programs were able to achieve a mean speedup of 3.97× on a 4-GPU machine over 1 GPU execution times. For all the programs, our overlap technique combined with the FOP scheme achieved a geometric mean speed up of 2.97× over 1 GPU execution time which is significantly better than the speed up of 1.53× achieved without it.

Size of a tile plays an important role in determining its computation time and the communication volume. In many communication schemes, the overhead of communication will be proportional to the tile size (eg: FO). In such cases, compute-copy overlap provides a way to offset these overheads up to a certain extent for varying tile sizes. This extent is determined by the actual computation time of the tile and by maximizing compute-copy overlap we can utilize the computation time to the fullest extent thereby maintain the same performance for different tile sizes.

Figure 5.4a provides a comparison of the execution times for `floyd` with FO scheme on a 2-GPU machine and varying tile sizes. For `floyd`, the volume of communication in FO scheme is exactly equal to the size of the tile. Hence the communication overhead keeps increasing as the tile sizes are increased. The FO bar shows the execution time without compute-copy overlap and we can clearly see that the execution time increases proportional to the tile size. The same is the case even with overlap though the extent of performance loss is smaller. However, with tile reordering, we see that the execution time remains the same for tile size from 1 up to 32 and then increases at 64. This implies

that tile reordering was able to completely hide the overhead of data movement up to tile size of 32 indicating its resilience to tile size selection.

Chapter 6

Related Work

To the best of our knowledge, there is no work in the literature which can be used as a direct comparison to ours. The closest one is that of Kim et al [21]. However, the implementation is not available for comparison. Hence, we provide only a detailed discussion of this work.

Kim et al [21] propose a runtime system that takes a OpenCL program written for a single device and automatically runs them on multiple GPUs on a single machine. In this scheme, the uppermost and lowermost iterations of the partitioned work-group are sampled at runtime to determine the array region accessed by that work-group. This is similar to convex bounding box technique but done at runtime. Such a convex bounding box can be a very large array region even when the actual accessed area is much smaller. In the worst case, this can be as large as the entire array. To ensure consistency, this

Framework	**Allocation granularity**	**Memory mgmt scheme**	**Manual / Auto**	**#devices**
Kim et al [21]	convex bounding box	virtual CPU buffer	automatic	multiple
StarPU [3]	user-provided	MSI-based coherency	manual	multiple
CGCM [19]	entire array	modified runtime libraries	automatic	single
DyManD [18]	entire array	modified runtime libraries	automatic	single
Lee et al [23]	entire array	live variable analysis	user-annotated	single
Pai et al [28]	x10CUDA Rail	compiler inserted checks	automatic	single
Pluto-CUDA [5]	entire array	none	automatic	single
PPCG [41]	entire array	none	automatic	single
OpenACC [26]	entire array	none	user-annotated	single
BBMM (our)	disjoint bounding boxes	Runtime memory manager	Automatic	multiple

Table 6.1: Existing data allocation and buffer management schemes

scheme first moves the entire convex array region to the CPU even if one only a elements within the region is actually updated. Such a data movement will involve significant coherency overheads due to limited PCIex bandwidth. Figures 2.3g and 2.3h give a good estimate of the inefficiencies inherent in this scheme in terms of data allocation sizes and coherency overheads.

OpenACC [26] is a directive-based framework for accelerating applications using GPUs. It provides a set of compute, data and control flow directives for executing parallel *for* loops on accelerators. From memory management aspect, OpenACC provides copyin and copyout clauses which can be used to transfer data in and out of the GPU memory. It also provides *deviceptr* clause which can be used to mark data as already allocated on a GPU and hence provides a basic facility to reuse data already on the GPU. However, when it comes to multi-GPU machines, OpenACC leaves the burden of array distribution and coherency on the programmer. Programmer has to explicitly perform the distribution using either OpenACC provided data clauses or use CUDA [11] for memory allocation and data transfers. Integrating BBMM into OpenACC framework can bridge this gap and enable automatic data scaling for affine loop nests.

StarPU [3] is a dynamic task creation and scheduling framework for heterogeneous systems. StarPU's strength lies in its ability to automatically schedule tasks on one or more compute devices. However, StarPU's support for data allocation is completely manual. The programmer has the responsibility of identifying data allocation granularity, task-to-data mapping and inter-task data dependences. These are done automatically in BBMM. StarPU synchronizes data at allocation size granularity using the MSI-based coherency protocol. Hence, an entire blocks will be synchronized even if a very small portion of it is actually written by the task. This will lead to significant inefficiencies. BBMM's techniques can be adapted into StarPU framework to complement its automatic scheduling schemes with automatic data allocation as well.

CUDA [11] 4.0 and higher provide an abstraction called Unified Virtual Addressing (UVA). UVA abstracts away from the programmer the actual location of data, whether on any of the GPUs or on the CPU. Though this provides a cleaner and easier API for

the programmer to perform data allocation and transfers, it still has to be done manually when dealing with arrays larger than a single GPU's memory (since UVA's allocation unit granularity for arrays is the entire array). If multiple regions of a large array are used by a kernel this task remains tedious. The same applies to inter-GPU data movement that should be done to keep the manually distributed array regions across multiple GPUs in sync. In addition, reuse of data across kernels scheduled on the same GPU will also have to be managed manually. All of this is automatically handled by our system. In fact, UVA could be internally used by BBMM to simplify its implementation.

Baskaran et al [4] propose a data allocation scheme in the context of local memory optimization. This algorithm first identifies non-intersecting data spaces and groups them together. Then, for each such group, it finds the convex union (convex hull) of all the points in the partition and allocates a single convex bounding box encapsulating the entire partition. This scheme only works if the access functions are already disjoint and can be determined to be so at compile time. However for cases as shown in figures 2.3c this will not be the case and will be put in the same partition. Unlike our algorithm, this approach does not try to perform any operations to create disjoint sets if they are not already so. The bounding box over the convex hull can be a very large array region even when the actual accessed area is much smaller.

Größlinger [14] proposes techniques for precise scratchpad management on GPUs. The technique applies in our data scaling context as well. Even though it reduces memory utilization in some cases when compared to bounding box based schemes, the cost of access functions computed through the barvinok library can become prohibitively high due to an explosion in the number of index checks that need to be done. This happens when there are multiple distinct access functions, and in these cases it does not appear to be a practical solution.

There are many other works which propose compile-time and runtime techniques to ease different aspects of programming GPU setups [5, 18, 19, 23, 24, 28, 34–36, 41, 42]. Most of these target single GPU setups and hence they allocate entire array on the GPU. Table 6.1 gives a brief summary of these works from the aspect of memory management.

Lee et al. [24] propose techniques to map C code with OpenMP annotations into thread blocks and threads in a CUDA kernel. With OpenMPC [23], they extend this work with proposals for new GPU specific constructs. These works target code generation for a single GPU for both regular and irregular data accesses. The shared data in OpenMP are mapped into the global memory on the GPU. For shared arrays, the entire array is allocated on the GPU. Our work can complement this work in terms of data allocation for array data and affine loop nests.

Jablin et al. [18, 19], describe CPU-GPU communication management and optimization systems - CGCM and DyManD. In both the works, data is allocated and moved at the granularity of an allocation unit which for an array is the entire array. The works does not address multi-GPU scenarios. However, these works handle both regular and irregular data whereas we only handle affine array data but more efficiently than these two works.

X10 [43] is a programming language for shared and distributed-memory systems that uses the Partitioned Global Address Space (PGAS) memory model. X10 provides basic primitives to distribute a set of computations across multiple *places* (processing units) each of which contain some data and perform *activities* that operate on the data. In this direction, X10 provides API for defining Array and Regions which can be distributed across places. It also provides set operations on these regions. X10 has a CUDA extension which provides users with the facility to utilize X10 features with CUDA. However, like CUDA and OpenCL, users have to manually perform data allocation, memory management and coherency handling using the basic API provided by the language.

Wolfe [42] describes the design of the PGI accelerator framework. PGI is a commercially available accelerator product from Portland group. However the product does not support automatic distribution of loops among multiple GPUs. To use multiple GPUs user has to explicitly create tasks and schedule them on different GPUs using the provided annotations. HMPP [34] is a commercially available accelerator programming framework from CAPS enterprise. HMPP has an OpenCL code generator in their framework. However to the best of our knowledge, HMPP does not support automatically distributing

loops across multiple GPUs.

Pai et al [28] propose a compiler assisted runtime coherence system that moves data between the CPU and GPU only when the data on either device is stale. The data is transferred at the granularity of a CUDA X10 Rail. With each rail a status information is maintained which tells whether the rail is stale or not. Only if the rail is stale, and it is being read by another device, then data is moved between devices. For arrays, a Rail is an entire array and hence allocation happens at the array granularity. Even in terms of coherency, the work suffers from the same shortcoming as that of CGCM in that even if one element in a Rail is written to, then the entire array has to be synchronized.

Pluto-CUDA [5] and PPCG [41] generate CUDA code from serial C code for a single GPU. However they both lack a sophisticated memory allocation scheme. The entire input array is allocated on each device.

Chapter 7

Conclusions

7.1 Summary

Multi-GPU machines are being increasingly used in HPC setups. However, manually programming these machines to extract the combined power of the GPUs remains a tedious, time consuming and error-prone task. In order to ease the effort involved in programming multi-GPU machines, it is necessary to solve various compiler and runtime challenges in an automatic, scalable, and efficient way. We have addressed one such challenge, that of data allocation and memory management, that was not previously addressed in an efficient way.

We presented a fully automatic data allocation and buffer management scheme for affine loop nests that allows parallel execution on multi-GPU machines. Through this scheme, data allocation, buffer management, and data transfers were all done at the granularity of (hyper)-rectangular regions of arrays (bounding boxes). Doing so allowed us to perform common set operations on these bounding boxes at runtime with negligible overhead. We used these operations to minimize data allocation sizes, the number of redundant allocations and data transfers, and exploit inter-tile reuse. We also presented the adaptation of an efficient inter-GPU data movement scheme that significantly reduced the coherency overhead. We further improved this with a technique to achieve maximum compute-copy overlap so that data movement overhead can be hidden within

the computation time.

On a 4-GPU machine, our scheme was able to achieve allocation size reductions of up to 75% when compared to existing schemes and allow excellent weak scaling. Our data movement scheme was able to significantly reduce the coherency overhead with the communication volume reduced by a factor of 11× to 83× over state-of-the-art, translating into a mean execution time speedup of 1.53×. We compared our automatically generated OpenCL codes with manually written multi-GPU OpenCL and OpenACC codes and found that they yielded performance of at least 88% of the performance of manual OpenCL codes and outperformed the OpenACC codes. We also demonstrated the potential of our scheme to swap-in and swap-out required data, by showing data scaling with applications on input sizes significantly greater than the combined memory size of all GPUs. All of these were achieved while incurring very low runtime overhead – of less than 0.1% of overall execution time.

Our work is suited for any compiler/runtime system targeting GPUs. Doing so will bridge the data allocation gap that currently exists in programming these systems.

7.2 Future work

- The core ideas and techniques used in our work are generic and can be easily extended for memory management in non-GPU setups such as distributed memory clusters, and heterogeneous setup consisting of both CPU and GPU devices.

- The principle of rectangular set operations can be extended to non-affine programs as well, if one can come up with a way to approximate the region of array accessed by the program into a hyper-rectangle. One example case that can be targeted is sparse-matrix computations.

- Our technique can be ideally integrated either into a OpenACC compiler or a tool that can automatically generate OpenACC directives for affine programs. This can further reduce the programmer effort to scale programs on multi-GPU systems.

References

[1] Vikram Adve and John Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 186–198, New York, NY, USA, 1998. ACM.

[2] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 126–138, New York, NY, USA, 1993. ACM.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In *Concurrency and Computation: Practice and Experience*, 2009.

[4] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 1–10, New York, NY, USA, 2008. ACM.

[5] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on*

Compiler Construction, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] Cédric Bastoul. Clan: The Chunky Loop Analyzer, 2005. The Clan User guide.

[7] Uday Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *ACM/IEEE Supercomputing (SC '13)*, Denver, Colorado, USA, November 2013. ACM.

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[9] Daniel Chavarría-Miranda and John Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 14–25, New York, NY, USA, 2005. ACM.

[10] M. Classen and M. Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *IEEE IPDPS*, April 2006.

[11] NVIDIA CUDA, 2011. http://developer.nvidia.com/object/cuda.html.

[12] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *The 22nd International Conference on Parallel Architectures and Compilation Techniques (ACM/IEEE PACT)*, Edinburgh, Scotland, 2013.

[13] NVIDIA Fermi Compute Architecture, 2010. http://www.nvidia.in/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[14] Armin Größlinger. Precise management of scratchpad memories for localising array accesses in scientific codes. In *Proceedings of the 18th International Conference on*

Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, CC '09, pages 236–250, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2002.

[16] J. L. Hennessy and D. A. Patterson. *Compute Architecture: A Quantitative Approach.* Elsevier, Fourth Edition.

[17] Integer Set Library, 2012. Sven Verdoolaege, An Integer Set Library for Program Analysis (http://www.ohloh.net/p/isl).

[18] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. Dynamically managed data for cpu-gpu architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 165–174, New York, NY, USA, 2012. ACM.

[19] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic cpu-gpu communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM.

[20] Khronos Group. http://www.khronos.org.

[21] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in opencl for multiple gpus. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 277–288, New York, NY, USA, 2011. ACM.

[22] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff. A hybrid

approach of openmp for clusters. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 75–84, New York, NY, USA, 2012. ACM.

[23] Seyong Lee and Rudolf Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[24] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[25] NVIDIA GPU Computing SDK, 2010. https://developer.nvidia.com/gpu-computing-sdk.

[26] OpenACC Application Programming Interface, 2012. http://www.openacc-standard.org/.

[27] OpenCL, 2011. http://www.khronos.org/opencl.

[28] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. Fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 33–42, New York, NY, USA, 2012. ACM.

[29] HPC opensource project, 2012. www.par4all.org.

[30] Portland Group Inc., 2012. http://www.pgroup.com/.

[31] PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[32] The polybench project, 2012. http://polybench.sourceforge.net.

[33] PolyLib - A library of polyhedral functions, 2010. http://icps.u-strasbg.fr/polylib/.

[34] François Bodin Romain Dolbeau, Stéphane Bihan. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2007.

[35] Fengguang Song and Jack Dongarra. A scalable framework for heterogeneous gpu-based clusters. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 91–100, New York, NY, USA, 2012. ACM.

[36] Fengguang Song, Stanimire Tomov, and Jack Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 365–376, New York, NY, USA, 2012. ACM.

[37] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: data orchestration and tuning for opencl devices. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, 2010.

[38] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[39] Sanket Tavarageri, Albert Hartono, Muthu Baskaran, Louis-Noel Pouchet, J. Ramanujam, and P. Sadayappan. Parametric Tiling of Affine loop nests. Technical report, 2013.

[40] The Top 500 Supercomputers. http://www.top500.org.

[41] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[42] Michael Wolfe. Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, New York, NY, USA, 2010. ACM.

[43] X10 programming language, 2013. http://x10-lang.org/.