Automatic Data Allocation and Buffer Management for Multi-GPU Machines

THEJAS RAMASHEKAR and UDAY BONDHUGULA, Indian Institute of Science

Multi-GPU machines are being increasingly used in high-performance computing. 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. Existing works that propose to automate data allocations for GPUs have limitations and inefficiencies in terms of allocation sizes, exploiting reuse, transfer costs, and scalability.

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 can perform at *runtime* standard set operations like union, intersection, and difference, finding subset and superset relations on hyperrectangular regions of array data (bounding boxes). It uses these operations along with some compiler assistance to identify, allocate, and manage data required by applications in terms of *disjoint* bounding boxes. This allows it to (1) allocate exactly or nearly as much data as is required by computations running on each GPU, (2) efficiently track buffer allocations and hence maximize data reuse across tiles and minimize data transfer overhead, and (3) and as a result, maximize 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 four-GPU machine with various scientific programs showed that BBMM reduces data allocations on each GPU by up to 75% compared to current allocation schemes, yields performance of at least 88% of manually written code, and allows excellent weak scaling.

Categories and Subject Descriptors: C.2.2 [**Programming Languages**]: Compilers, Optimization

General Terms: Compilers, Algorithms, Scalability

Additional Key Words and Phrases: GPU, memory management, data scaling, weak scaling, OpenCL, polyhedral model

ACM Reference Format:
Ramashekar, T. and Bondhugula, U. 2013. Automatic data allocation and buffer management for multi-GPU machines. ACM Trans. Architec. Code Optim. 10, 4, Article 60 (December 2013), 26 pages.
DOI: http://dx.doi.org/10.1145/2544100

1. INTRODUCTION

Multi-GPU machines are becoming commonplace in High-Performance Computing (HPC) setups. Such machines generally consist of one or more multicore CPUs and several GPGPUs connected on the PCI express interface. Each of the GPUs has its own memory and does not share the address space with the host CPU or with other GPUs. These machines are used both as standalone workstations to run computations

---

New article, not an extension of a conference paper.
Authors' addresses: T. Ramashekar, Indian Institute of Science - Computer Science and Automation Dept. of Computer Science and Automation Indian Institute of Science, Malleshwaram Indian Institute of Science, Malleshwaram, Bangalore, Karnataka 560012 India; email: thejascr@gmail.com.

DOI: http://dx.doi.org/10.1145/2544100

on medium to large data sizes (tens of gigabytes) and as a node in a CPU-multi-GPU cluster handling very large data sizes (hundreds of gigabytes to a few terabytes).

A significant body of scientific applications that utilize multi-GPU machines contain computations inside affine loop nests; that is, loop nests that have affine bounds and the array access functions in the computation statements are affine. These include stencils, linear algebra kernels, dynamic programming codes, and data-mining applications. 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 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 data transfers that are required to maintain coherence between GPUs; and
(5) Aggregate the final results from the GPUs onto the host CPU.

Data allocation, buffer management, and coherency handling are a critical part of these 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 [Augonnet et al. 2009; CUDA 2011; OpenACC 2012] or automatic schemes [Kim et al. 2011] 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 article, we present the design of such an automatic multi-GPU memory manager that embeds itself into steps (2), (4), and (5) mentioned previously and performs those tasks efficiently on behalf of the application.

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 a 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; and
—To ensure that the overhead of achieving these tasks does not adversely affect the overall execution time of the program.

One technique that has been used in certain memory management contexts in the literature is that of *bounding boxes* [Kim et al. 2011; Baskaran et al. 2008]. A bounding box of an array, for a given tile, is the smallest hyperrectangle that encapsulates all of 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 article, we exploit the potential of bounding boxes for memory management far beyond their current usage in the literature, with the following key insights:

—Bounding boxes can be subjected to standard set operations like union, intersection, difference, and so forth 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 these 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 intertile data reuse and minimize CPU-GPU data movement overhead. All data transfers needed to maintain inter-GPU coherency are performed in terms of bounding boxes, thereby exploiting the architectural benefits provided by GPUs. The runtime operations incur negligible overhead.

The contributions of this article are as follows:

—The design of a fully automatic, efficient, and highly scalable memory manager for affine loop nests on multi-GPU machines;
—A compiler-assisted runtime algorithm to store and manage accessed array data as a set of *disjoint* bounding boxes;
—Efficient schemes to maximize intertile data reuse and minimize CPU-GPU data movement overhead; and
—Experimental evaluation demonstrating scalability on applications with large dataset sizes on a multi-GPU machine.

The rest of this article is organized as follows. Section 2 gives examples to motivate the contributions of the article. Section 3 gives a brief introduction to the polyhedral framework. Section 4 gives the definition and key insights regarding bounding boxes. Section 5 gives a high-level overview of BBMM. Sections 6 and 7 describe the core data allocation and buffer management algorithms. Section 8 describes the inter-GPU coherency scheme used in BBMM. Section 9 gives the overall structure of the generated host and kernel code. Section 10 provides implementation details, while experimental evaluation is provided in Section 11. Section 12 discusses related work, and Section 13 provides conclusions.

2. MOTIVATING EXAMPLE

We first provide a brief discussion of the general structure of the code that runs on a multi-GPU machine. We then provide examples that motivate contributions of this work.

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 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

```
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 }
```

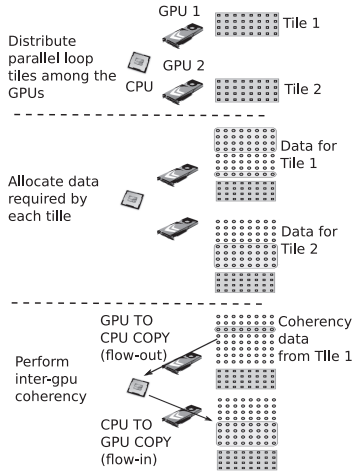


Fig. 1. General structure of an affine loop nest for a multi-GPU machine.

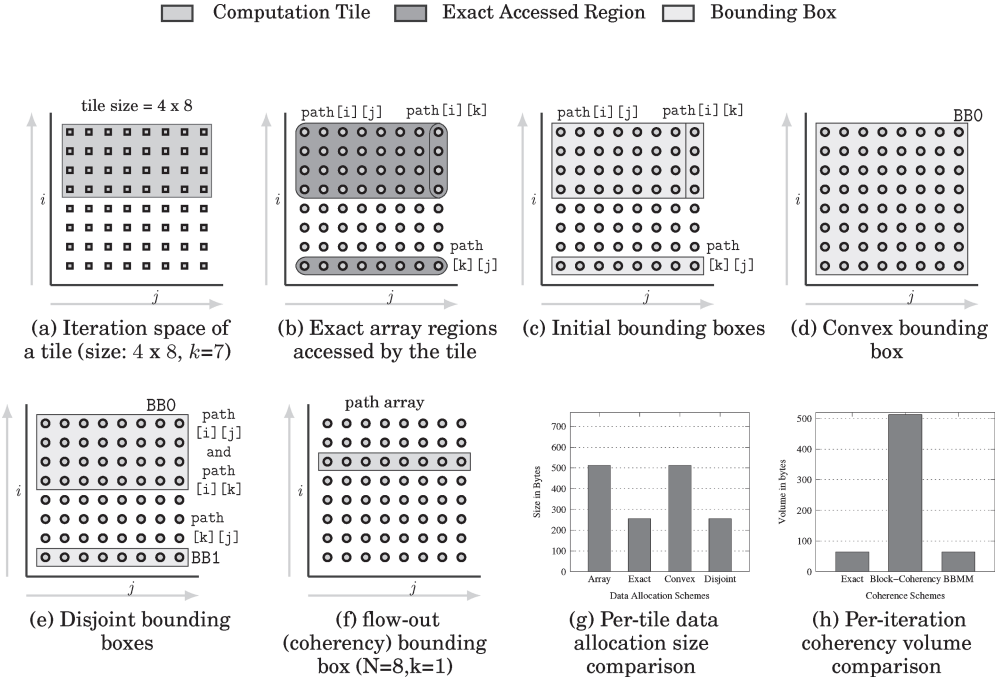
repeated for each of them. The execution begins on the host CPU with the outermost serial loop (if it exists) as shown in line 11. For each surrounding serial loop iteration, the tiles of the parallel band are distributed among the available GPUs (line 13). For each tile assigned to a GPU, the data allocation 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 8). 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 coherence 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

The `floyd-warshall` algorithm computes the shortest path between every pair of vertices in a weighted directed 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 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]`,`path[k][j]`. `floyd-warshall` has nonuniform array access patterns. Hence, depending on the value of k, these access functions access regions of array that either intersect or are disjoint.

```
for (k=0; k<N; k++) /* outer serial loop */
  for (i=0; i<N; i++) /* outermost 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];
```

Fig. 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) flow-out (coherency) bounding box (N=8,k=1)
(g) Per-tile data allocation size comparison
(h) Per-iteration coherency volume comparison

Fig. 3. Per-tile data allocation, coherency, and reuse exploitation on `floyd-warshall`.

Figure 3(a) shows the iteration space for Floyd-Warshall with N set to 8 and k set to 7. A tile of size 4×8 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 3(b)).

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 3(c). Some works in the literature [Kim et al. 2011; Baskaran et al. 2008] 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 3(d), even though the actual region accessed by the tile is much smaller. BBMM, however, performs set operations such as union, intersection, difference, and so forth on these initial bounding boxes and allocates the data required by the tile in terms of *disjoint* bounding boxes. Figure 3(e) 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 3(g) and equal to the exact array regions accessed by the tile.

Inter-GPU coherency: Dependence analysis of `floyd-warshall` can show that, in 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 the $k-1^{th}$ 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 3(f) needs to be synced with other GPUs. However, coherency schemes used in the closely related works [Kim et al. 2011] transfer the entire data allocation block, which contains 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 needs to be synchronized), and almost the entire array in the worst case, that is, when k is 6. Figure 3(h) 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.

Exploiting intertile 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 that 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.

3. BACKGROUND ON POLYHEDRAL MODEL

In this section, we give a brief introduction to the polyhedral model that is used in the compile-time component of BBMM.

The polyhedral model provides a framework to compactly capture the execution sequence 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 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 an n-tuple, where n is the dimensionality of the iteration space. In an affine loop nest, all array subscript expressions in the computation statements have to be affine functions of the outer loop iterators and program parameters. 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, and difference on these polyhedra [Polylib 2010; ISL 2012]. For more details on the polyhedral framework, readers are referred to Bastoul [2005].

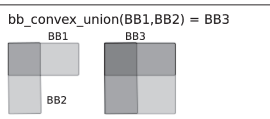
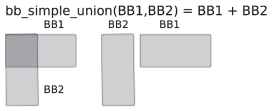
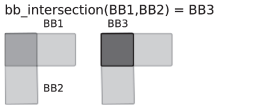
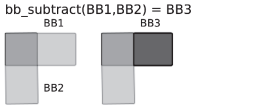
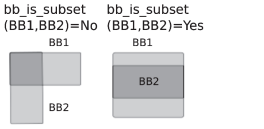
4. BOUNDING BOXES AND SET OPERATIONS

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

Bounding box: BBMM allocates and manages data in terms *hyperrectangles*. A hyperrectangle is an n-dimensional counterpart of a rectangle; that is, it has two parallel faces for each dimension that bound the rectangle along that dimension. The smallest hyperrectangle encapsulating a point set is called the *bounding box* of that set. In our context, a bounding box is the smallest hyperrectangle encapsulating the elements of a multidimensional array that are read or written through an access function in a computation statement of a program.

Set operation on bounding boxes: The integer points inside the bounding boxes can be subjected to common set operations like union, intersection, and difference, finding

Table I. Set Operations of BBMM and Their Overhead

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	

subset and superset relations. Table I lists and illustrates these operations. Using them, BBMM can perform various memory optimizations such as refining compiler- generated bounding boxes, exploiting intertile reuse, minimizing data movement overhead, and so forth. 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, that is, as `a[i-`lb_i`][j-`lb_j`]`, where lb_i and lb_j are lower bounds of the bounding box along dimensions 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

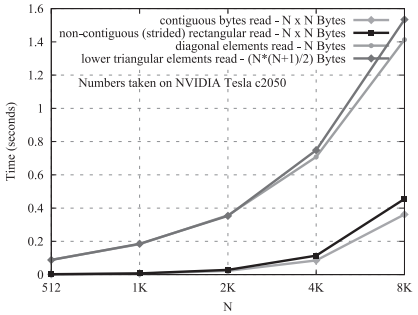


Fig. 4. Data transfer time for various access shapes.

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 noncontiguous (strided) rectangular transfers of data between the CPU and GPU are almost as efficient as transferring contiguous bytes (Figure 4). This may be due to internal packing of noncontiguous elements before performing a single DMA to transfer it to destination buffers on the CPU. The same is done by the runtime driver during CPU to GPU transfers. These noncontiguous 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) *Nonscalability of precise data allocation techniques*: Größlinger [2009] 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 Section 12), thus making it impractical.

5. OVERVIEW OF BBMM

In this section, we give a high-level view of the various components of BBMM. Figure 5 shows the interworking of the input, the compile time, and the runtime components.

Input to BBMM-computation tile: BBMM expects the loop nests to be parallelized and tiled to suit GPU architectures. 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.

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 refines 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.

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:

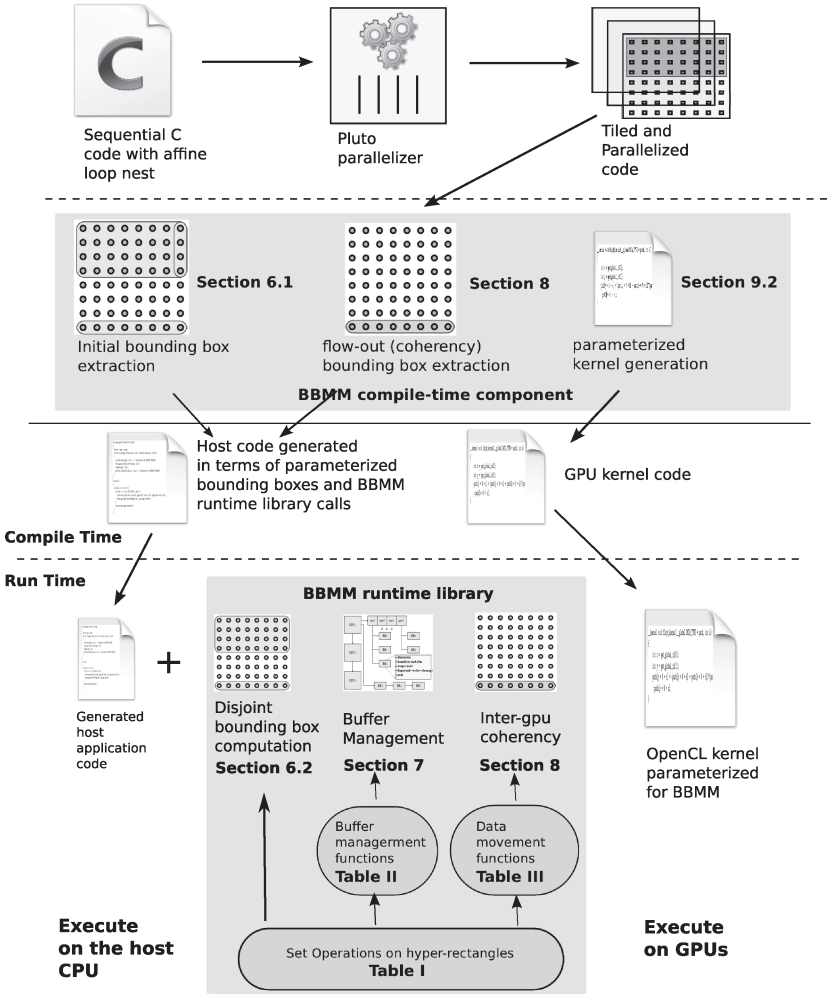


Fig. 5. High-level overview of BBMM.

—Hyperrectangular set operations used to refine the bounding boxes and thereby minimize memory allocation;
—Buffer management functions that track the bounding boxes on a per-GPU basis and performs various memory optimizations; and
—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.

6. 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

ALGORITHM 1: `extract_initial_bounding_boxes()`

Input: Computation tile $\vec{t}$, Array a
1 $S_a^{init} = \phi$
2 **for each** *read or write access function* f_a^i **do**
3 dp_a^i = get_data_polyhedron(f_a^i)
4 bb_a^i = get_bounding_box(dp_a^i)
5 add bb_a^i to S_a^{init}
6 **Output**: S_a^{init}, the set of initial bounding boxes

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.

6.1. Initial Bounding Box Extraction at Compile Time

Bounding box extraction 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, and
`get_bounding_box()`: computes the smallest hyperrectangle encapsulating all the points in the data space.

The set of bounding boxes returned is 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 overallocations because 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.

6.2. 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 I to subtract out the portions that 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.

6.3. Example

Figure 3 illustrates the data allocation scheme for `floyd-warshall`. Figure 3(a) shows the iteration space of a single tile. For illustration purposes, we have chosen $N = 8$ and $k = 7$. Figure 3(c) 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 3(e) shows the initial bounding boxes as they look 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 3(e) shows the result of

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

running Algorithm 2 on the initial bounding box list. For the chosen tile, `path[i][k]` is a subset of `path[i][j]`, whereas `path[k][j]` is disjoint from both. Hence, the algorithm returns two disjoint bounding boxes BB0 and BB1 as shown.

6.4. 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 that 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 might seem 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; that is, they take the same control flow path, resulting in no performance loss.
—In order to incur a performance loss due to warp divergence, an access function should both be nonuniform (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 prohibitive.

In Section 11, we provide the results that support the aforementioned reasoning.

7. 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, reducing duplicate allocations, maximizing intertile data reuse, and providing capability to work with data sizes larger than the available GPU memory.

7.1. Design Overview

Figure 7 shows the design of the buffer management component. For each GPU, the buffer manager maintains two lists of bounding boxes: (1) in-use list and (2) unused list. The ones that are currently being read or written by a tile will be in the in-use 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 the unused list only if its usage count is zero. The usage count will be important

Table II. Functions Provided by the Buffer Manager

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 intradevice transfers if possible
bb_cleanup()	De-allocates bounding boxes that are no longer required

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 or written and what all needs to be cleaned up. The memory manager provides various inexpensive runtime functions on the bounding boxes. Table II lists the important ones.

7.2. Intertile Data Reuse

Using hyperrectangles 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 the 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.

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 intradevice 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 intradevice copy and the intersecting area is subtracted out from it. The process is repeated for all bounding boxes already present on the device. The leftover portion is initialized with the data from the CPU.

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 they are allocated. The `bb_cleanup()` function performs this task.

7.3. Freeing Up Space on a GPU—Box-in and Box-Out

The function `bb_alloc()` can free up 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. The 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 sizes much larger than the *combined* memory sizes of all GPUs. However, box-out and box-in operations are

highly expensive since they involve significant data transfers. Hence, for this feature to be of use, the tiles for which we are freeing space need to have sufficient computation 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 that had an outer serial loop performed worse because a single parallel tile within each outer serial iteration did not have a sufficient compute-to-copy ratio. The results and detailed analysis for this feature are presented in Section 11.

8. INTER-GPU COHERENCY

GPUs in a multi-GPU machine do not share address space. Hence, if tiles distributed across different GPUs access the 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 the 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 of that element (if they exist) 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.

8.1. 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.

8.2. 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 (8GBps). 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 [Amarasinghe and Lam 1993; Adve and Mellor-Crummey 1998; Chavarría-Miranda and Mellor-Crummey 2005; Classen and Griebl 2006; Bondhugula 2013; Kwon et al. 2012; Dathathri et al. 2013]. In this article, we use the state-of-the-art data movement scheme for distributed-memory scenarios proposed by Dathathri et al. [2013]. 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

```
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      // initial bounding box for path[i][j]
7      struct bounding_box * bb0 = bb_alloc(2);
8      bb0->array = array;
9      bb0->bp[0].lb = (TILE_SIZE * t3) + (0);  // lower bound along dimension 0
10     bb0->bp[0].ub = (TILE_SIZE * t3) + ((TILE_SIZE-1)); // upper bound along dimension 0
11     bb0->bp[1].lb = 0; // lower bound along dimension 1
12     bb0->bp[1].ub = +1*N-1; // upper bound along dimension 1 (parametric on N)
13     add_to_list(&init_bb_list, bb0);
14     // similar code as above for path[i][k] and path[k][j]...
15     ..
16     return init_bb_list;
17 }
```

Fig. 6. Compile-time-generated function that returns initial bounding boxes for a tile.

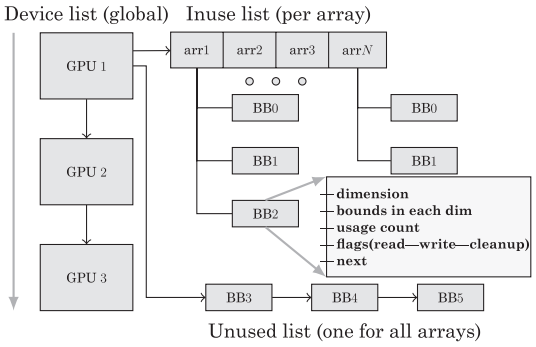


Fig. 7. Buffer management component of BBMM.

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. For details of this scheme, the reader is referred to Dathathri et al. [2013]. We describe the adaptation of this scheme and how we extend the idea of bounding boxes for intertile data transfers.

Rectangular transfers and flow-out bounding boxes: Since rectangular transfers (both contiguous and noncontiguous) are efficiently supported on the GPUs (Figure 4), 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 enable 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 6 is generated.

Table III. Functions Provided by the Data Movement Component

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

Data movement orchestration at runtime: The BBMM runtime library contains the data movement orchestration component. At runtime, three distinct data transfers are performed. Table III 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. The CPU always has the latest copy of the flow-out sets so that a future tile depending on a flow-out set can be provided the required data from the CPU itself. When the exact flow-out sets are nonrectangular, 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 all of the 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 nonrectangular, 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 have to be copied onto the destination bounding boxes.

9. OVERALL STRUCTURE OF THE GENERATED CODE

We now describe how all components of BBMM work together as part of the generated host and kernel code.

9.1. Structure of Generated Host Code

The overall structure of the host code for a single affine loop nest is shown in Algorithm 4. In a program with multiple independent affine loop nests, this structure is repeated for each of them. The execution begins from the outer serial loops. For each iteration of this loop, the set of parallel tiles is distributed among the available GPUs. This can be a one-dimensional or multidimensional block or block-cyclic distribution. The code following the distribution of tiles is executed in the context of a worker thread that 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 its 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 are complete, the intertile data transfers are performed using the functions described in Table III. Following this, the usage count of the bounding boxes is decremented and any bounding box that has a usage count of zero is moved to the unused list to be either reused or freed up for the next iteration. At the end of the outer serial loop iteration, the bounding boxes are marked for cleanup using `bb_cleanup()`.

ALGORITHM 4: `Structure of generated host code for a single affine loop nest`

```
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)
```

9.2. Structure of the Parameterized GPU Kernel

GPU kernel generation is an orthogonal problem to ours, and works such as Baskaran et al. [2010], Par4All [2012], and Verdoolaege et al. [2013] 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 9 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 are set to appropriate disjoint bounding box buffers using the `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 common case), the macro dereferences the buffer pointer associated with the bounding box using indices that are decremented by offsets of that bounding box. If the access function is split, then the macro checks each bounding box's bounds to determine which bounding box the index being accessed belongs to, and that buffer is dereferenced with the appropriate indices.

Algorithm 3: `bb_readin()`

Input: Device dev, Array a, Bounding box bb_a^{new}
for each *bounding box* bb_a^{dev} *on the device* **do**
  $bb_a^{intersect}$ = bb_intersection(bb_a^{new}, bb_a^{dev})
  if $bb_a^{intersect}$ *is not empty* **then**
    /* part of the data required for this bounding box is already present on the device */
    intra_device_copy(bb_a^{new}, bb_a^{dev}, $bb_a^{intersect}$)
    bb_a^{new} = bb_subtract(bb_a^{new}, $bb_a^{intersect}$)
if bb_a^{new} *is not null* **then**
  /* some portion of the bounding box is not yet read in. copy this data from cpu */
  cpu_to_gpu_copy(bb_a^{new})

Fig. 8. Algorithm for `bb_readin()`.

```
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  }
```

Fig. 9. General structure of the parameterized GPU kernel.

10. 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 [Bondhugula et al. 2008]. 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 tile sizes automatically is an orthogonal problem. Doing this, in part, requires parametric tiling, which is part of ongoing research in the polyhedral area [Tavarageri et al. 2013]. Hence, BBMM currently requires the user to provide the tile sizes at compile time. 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 6, 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 article, 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, and
—Tiles across different (independent) parallel loop nests are executed sequentially in program order across different kernel calls.

Table IV. Programs Used for Evaluation
A: number of arrays. B: maximum dimensionality of arrays. C: bounding box type chosen by our algorithm. D: maximum number of bounding boxes for any array. E: subsumed bounding boxes present? F: BBMM runtime overhead as a percentage of overall execution time.

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

We placed these restrictions to focus our experimentation on the efficiency and effectiveness of the key aspects 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.

11. EXPERIMENTAL EVALUATION

11.1. Setup and Benchmarks

The experiments were run on an Intel Xeon multicore server consisting of 12 Xeon E5645 cores (two-way SMP of hex-core) running at 2.4GHz. The server includes three NVIDIA Tesla C2050 and one NVIDIA Tesla K20 graphics processors connected on the PCI-express bus. The Tesla C2050s have 2.5GB of global memory each and the K20 has 5GB of global memory. For our experiments, we chose to limit the memory usage of K20 to 2.5GB to maintain uniformity. We thus have a combined GPU memory size of 10GB. NVIDIA driver version 304.64 supporting OpenCL 1.1 was used as the OpenCL runtime.

The programs for evaluating BBMM were drawn from several benchmarks: Polybench [2012], Pochoir [Tang et al. 2011], and the NVIDIA GPU SDK [2010]. The main criterion 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, nonuniform, embarrassingly parallel (EP)) and different array dimensionalities. The selected programs are listed in Table IV.

11.2. Evaluation Parameters and Results

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

11.2.1. Overhead of the Runtime Library. This is an important parameter that measures the execution overhead of BBMM runtime library functions, giving insight into the cost of the various bounding box set operations and the memory management functions listed in Tables I and II. 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:

$$\begin{aligned} total\_time &= memory\_mgmt\_time + compute\_time + flowout\_time \\ &\quad + flowin\_time + writeout\_time \\ overhead\_percentage &= (memory\_mgmt\_time/total\_time) * 100 \end{aligned}$$

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

11.2.2. Performance of Programs with Data Scaling. Data scaling refers to the scenario when dataset sizes of a program are increased in the same way as the combined memory size of the processing elements (GPUs in this case); that is, 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. 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, that is, 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.

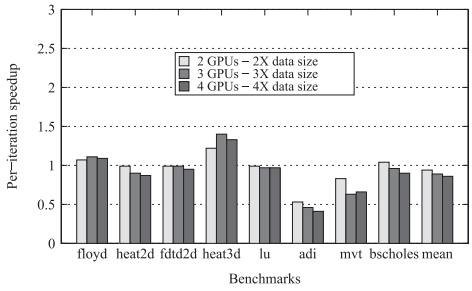
Figure 10(a) shows the data scaling results on two, three, and four GPUs. For every additional GPU added to the experiment, the data size is increased proportionally so that the data size per GPU is constant. For all programs except `adi`, we see that the per-iteration speedup is around one with a geometric mean of 0.94, indicating near-ideal data scaling. However, for `adi`, we see a significant slowdown with increase in the number of GPUs. This is because `adi` has a large amount of interdevice data movement at the end of each 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.

11.2.3. 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.

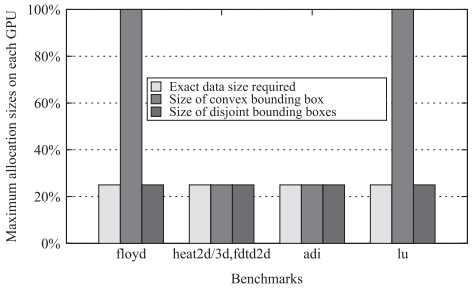
Figure 10(b) shows the reduction in data allocation sizes as a result of using disjoint bounding boxes. For `floyd` and `lu`, which have nonuniform data access patterns, using disjoint bounding boxes greatly reduces the allocation sizes—up to 75% on our 4 GPU machine as compared to convex bounding boxes. However, for the class of programs such as stencils and linesweep (`heat2d`, `heat3d`, `fdtd2d`, `adi`) that have uniform accesses, the difference in sizes is negligible. Also, for all the programs, the allocation sizes due to disjoint bounding boxes equal the exact required data sizes (computed manually). This shows that BBMM's automatically generated data sizes for these programs are as good as those with any possible manual programming effort.

11.2.4. Benefits of Intertile Data Reuse. This gives insight into the extent of 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.

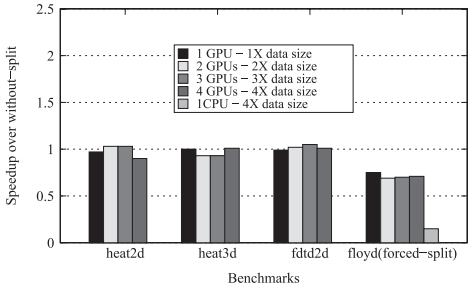
Figure 10(d) shows the speedup obtained with intertile data reuse enabled. The baseline is the execution time of the programs without enabling this optimization.



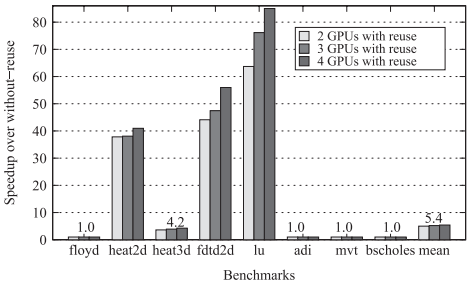
(a) Performance with data-scaling



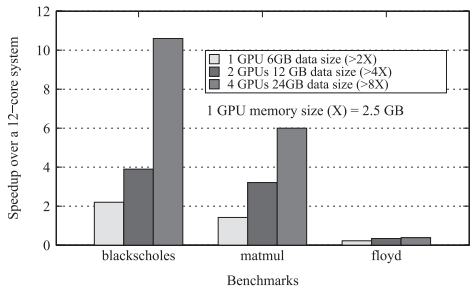
(b) Allocation size comparison between exact, convex and disjoint bounding box schemes



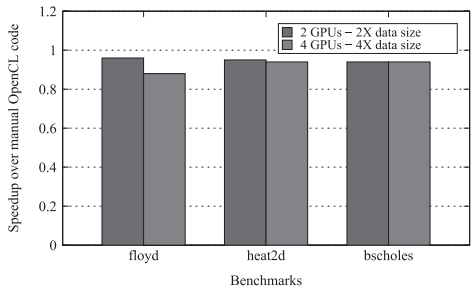
(c) Performance with access function splits as compared to without-split



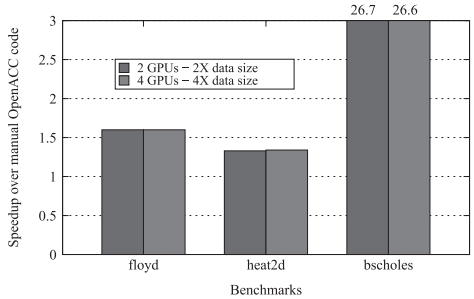
(d) Speedup with inter-tile reuse as compared to without-reuse



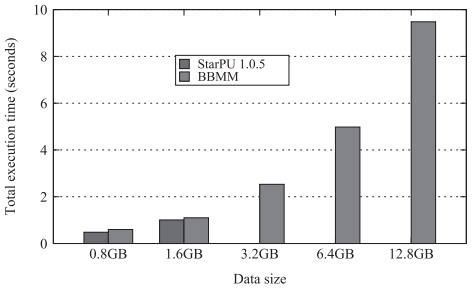
(e) Speedup with box-in and box-out over a 12-core system



(f) Performance normalized to manually written multi-GPU OpenCL code



(g) Performance normalized to manually written multi-GPU OpenACC code



(h) Comparison with StarPU for `mvt` on 1 GPU

Fig. 10. Results of various experiments.

Exploiting intertile data reuse yielded a mean speedup of 5.4 over the without-reuse case. `heat2d`, `heat3d`, `lu`, and `fdtd2d` have tiles whose bounding boxes are already fully subsumed inside bounding boxes of other tiles. When intertile data reuse is disabled, these subsumed bounding boxes are also allocated separately. The flow-out sets now have to be copied to these as well, which results in a 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.

11.2.5. 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`) that have uniform dependences undergo a split for some of their access functions. For programs with nonuniform 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—this is the worst-case scenario. Hence, in this case, there is a conditional check for every array access. This gives insight into the worst-case performance of the generated kernel.

Figure 10(c) 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 6.4, we see that stencils do not have any performance degradation even when the bounding boxes are split. However, for `floyd`, we see a 30% 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 30% degradation, we note that the performance on a GPU is much better than that of a tiled and parallelized version of `floyd` running on the 12-core system.

11.2.6. Benefit of Box-in and Box-out. As explained in Section 7.3, 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` 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 multicore 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 10(e) 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 an 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 article, we do not present results with such a tiling.

11.2.7. Comparison with Manual Code. We provide a comparison of BBMM's automatically generated memory management code with manually written OpenCL [2011] and OpenACC [2012] 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 the StarPU [Augonnet et al. 2009] 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 a nonuniform access pattern, `heat2d` for its uniform access pattern and intertile 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 characteristics present in other programs of our test suite. The codes were written with the following optimizations:

—Computations were manually distributed equally among the GPUs;
—Data allocation sizes for each tile of computation were theoretically minimum;
—Volume of data moved for coherency was the theoretical minimum; and
—Reuse exploitation was theoretically maximum; that is, no data that had already been on the GPU was reallocated or reinitialized.

Figure 10(f) shows the relative execution time of BBMM as compared to manually written OpenCL code on two and four 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 the 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 and (2) additional memory accesses present in the generated kernel. However, considering the significant programming effort involved in writing the manual code, we find this slight slowdown acceptable.

Comparison with manually written OpenACC codes: OpenACC [2012] 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 subarrays 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 2012 compiler version 12.10, supporting the OpenACC 1.0 specification, was used to compile the codes. As in the case of OpenCL, the allocation sizes and coherency volume were kept to the theoretical minimum. Again, we chose `floyd`, `heat2d`, and `blackscholes` as representative programs for nonuniform, uniform, and embarrassingly parallel applications.

Figure 10(g) shows the comparison of the execution times of BBMM and OpenACC on two and four 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 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 10(h) 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 whenever the data size was less than the GPU memory size, performance of BBMM's automatically generated code was on par with the manual version of the code. Beyond the GPU memory 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 12.8GB ($>5\times$ the single GPU memory).

12. RELATED WORK

To the best of our knowledge, there is no work in the literature that can be directly compared with ours. The closest one is that of Kim et al. 2011. However, the implementation is not available for comparison. Hence, we only compare it through a discussion here.

Kim et al. [2011] propose a runtime system that takes an OpenCL program written for a single device and automatically runs it on multiple GPUs on a single machine. In this scheme, the uppermost and lowermost iterations of the partitioned workgroup are sampled at runtime to determine the array region accessed by that workgroup. This is similar to the 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 (e.g., `floyd` and `lu` will have this problem due to the nonuniform nature of accesses in them). BBMM overcomes this problem by computing and managing disjoint bounding boxes. To ensure coherency, their scheme first moves the entire convex array region to the CPU even if only a few elements within the region are actually updated, which can lead to significant data movement overhead (among our chosen programs, all except `mvt` and `blackscholes` will be negatively affected by this technique). BBMM however, uses near-precise and efficient data movement techniques, thereby minimizing this overhead. Figures 3(g) and 3(h) give a good estimate of the inefficiencies inherent in this scheme in terms of data allocation sizes and coherency overheads.

OpenACC [2012] 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 a memory management aspect, OpenACC provides copy-in and copy-out clauses, which can be used to transfer data in and out of the GPU memory. It also provides *deviceptr* clauses, which can be used to mark data as already allocated on a GPU and hence provide 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. A programmer has to explicitly perform the distribution using either OpenACC-provided data clauses or CUDA [2011] for memory allocation and data transfers. Integrating BBMM into the OpenACC framework can bridge this gap and enable automatic data scaling for affine loop nests.

StarPU [Augonnet et al. 2009] 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 intertask data dependences. These are done automatically in BBMM. StarPU synchronizes data at allocation size granularity using the MSI-based coherency protocol. Hence, an entire block will be synchronized even if a very small portion of it is actually written by the task. This will lead to significant data movement overhead.

CUDA [2011] 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

Table V. Existing Data Allocation and Buffer Management Schemes

Framework	Allocation granularity	Memory mgmt scheme	Manual/Auto	#devices
[Kim et al. 2011]	convex bounding box	virtual CPU buffer	automatic	multiple
[Augonnet et al. 2009]	user provided	MSI-based coherency	manual	multiple
[Jablin et al. 2011]	entire array	modified runtime libraries	automatic	single
[Jablin et al. 2012]	entire array	modified runtime libraries	automatic	single
[Lee and Eigenmann 2010]	entire array	live variable analysis	user annotated	single
[Pai et al. 2012]	x10CUDA Rail	compiler-inserted checks	automatic	single
[Baskaran et al. 2010]	entire array	none	automatic	single
[Verdoolaege et al. 2013]	entire array	none	automatic	single
[OpenACC 2012]	entire array	none	user annotated	single
BBMM (ours)	disjoint bounding boxes	runtime memory manager	automatic	multiple

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 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. [2008] propose a data allocation scheme in the context of local memory optimization. This algorithm first identifies nonintersecting 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 Figure 3(c), 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 [2009] 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 that propose compile-time and runtime techniques to ease different aspects of programming GPU setups [Baskaran et al. 2010; Verdoolaege et al. 2013; Lee et al. 2009; Lee and Eigenmann 2010; Wolfe 2010; Jablin et al. 2011, 2012; Song and Dongarra 2012; Song et al. 2012; Pai et al. 2012]. Most of these target single GPU setups and hence they allocate an entire array on the GPU. Table V gives a brief summary of these works from the aspect of memory management.

13. CONCLUSIONS

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 addressed one such challenge, that of data allocation, 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 and the number of redundant allocations and data transfers and to exploit intertile reuse. On a four-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. 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 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.

REFERENCES

Adve, V. S. and Mellor-Crummey, J. M. 1998. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*. 186–198.

Amarasinghe, S. P. and Lam, M. S. 1993. Communication optimization and code generation for distributed memory machines. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*. 126–138.

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

Baskaran, M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. 2008. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*.

Baskaran, M. M., Ramanujam, J., and Sadayappan, P. 2010. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction 2010*.

Bastoul, C. 2005. Clan: The Chunky Loop Analyzer user guide. http://icps.u-strasbg.fr/~bastoul/development/clan/docs/clan.pdf.

Bondhugula, U. 2013. Compiling affine loop nests for distributed-memory parallel architectures. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'13)*.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. 2008. A practical automatic polyhedral program optimization system. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*.

Chavarría-Miranda, D. and Mellor-Crummey, J. 2005. Effective communication coalescing for data-parallel applications. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*.

Classen, M. and Griebl, M. 2006. Automatic code generation for distributed memory architectures in the polytope model. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'06)*.

CUDA 2011. NVIDIA CUDA programming model. http://developer.nvidia.com/object/cuda.html.

Dathathri, R., Reddy, C., Ramashekar, T., and Bondhugula, U. 2013. Generating efficient data movement code for heterogeneous architectures with distributed memory. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*.

Grösslinger, A. 2009. Precise management of scratchpad memories for localising array accesses in scientific codes. In *Compiler Construction*. 236–250.

ISL 2012. Integer Set Library. Sven Verdoolaege, An Integer Set Library for Program Analysis.

Jablin, T. B., Jablin, J. A., Prabhu, P., Liu, F., and August, D. I. 2012. Dynamically managed data for CPU-GPU architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'12)*.

Jablin, T. B., Prabhu, P., Jablin, J. A., Johnson, N. P., Beard, S. R., and August, D. I. 2011. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*.

Kim, J., Kim, H., Lee, J. H., and Lee, J. 2011. Achieving a single compute device image in openCL for multiple GPUs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*.

Kwon, O., Jubair, F., Eigenmann, R., and Midkiff, S. 2012. A hybrid approach of openmp for clusters. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*.

Lee, S. and Eigenmann, R. 2010. Openmpc: Extended openMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE Supercomputing Conference*.

Lee, S., Min, S.-J., and Eigenmann, R. 2009. 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)*.

NVIDIA GPU Computing SDK 2010. NVIDIA GPU Computing SDK.

OpenACC 2012. OpenACC Application Programming Interface.

OpenCL 2011. Open Computing Language. http://www.khronos.org/opencl/.

Pai, S., Govindarajan, R., and Thazhuthaveetil, M. J. 2012. 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)*.

Par4All 2012. HPC open source project - Par4All. http://www.par4all.org.

PGI 2012. Portland Group Inc. http://www.pgroup.com/.

Polybench 2012. The polybench project.

Polylib 2010. PolyLib - A library of polyhedral functions.

Song, F. and Dongarra, J. 2012. A scalable framework for heterogeneous gpu-based clusters. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*.

Song, F., Tomov, S., and Dongarra, J. 2012. 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)*.

Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. 2011. The Pochoir stencil compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. 117–128.

Tavarageri, S., Hartono, A., Baskaran, M., Pouchet, L.-N., Ramanujam, J., and Sadayappan, P. 2013. *Parametric Tiling of Affine Loop Nests.* Technical Report.

Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., and Catthoor, F. 2013. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim. 9,* 4, 54:1–54:23.

Wolfe, M. 2010. Implementing the PGI accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics (GPGPU'10)*.

Received June 2013; revised August 2013; accepted November 2013