Overlapping Data Transfers with Computation on GPU with Tiles

Burak Bastem*, Didem Unat*, Weiqun Zhang†, Ann Almgren†, John Shalf†
*Koç University, Istanbul, Turkey
{bbastem, dunat}@ku.edu.tr
†Lawrence Berkeley National Laboratory, CA, USA
{weiqunzhang, asalmgren, jshalf}@lbl.gov

***Abstract*—GPUs are employed to accelerate scientific applications however they require much more programming effort from the programmers particularly because of the disjoint address spaces between the host and the device. OpenACC and OpenMP 4.0 provide directive based programming solutions to alleviate the programming burden however synchronous data movement can create a performance bottleneck in fully taking advantage of GPUs. We propose a tiling based programming model and its library that simplifies the development of GPU programs and overlaps the data movement with computation. The programming model decomposes the data and computation into tiles and treats them as the main data transfer and execution units, which enables pipelining the transfers to hide the transfer latency. Moreover, partitioning application data into tiles allows the programmer to still take advantage of GPU even though application data cannot fit into the device memory. The library leverages C++ lambda functions, OpenACC directives, CUDA streams and tiling API from TiDA to support both productivity and performance. We show the performance of the library on a data transfer-intensive and a compute-intensive kernels and compare its speedup against OpenACC and CUDA. The results indicate that the library can hide the transfer latency, handle the cases where there is no sufficient device memory, and achieves reasonable performance.**

I. Introduction

Heterogeneous processors that combine accelerators and general-purpose multicores are no longer considered to be an exotic architecture but constitutes a large fraction of Top500 systems [1]. GPU-based heterogeneous systems have provided high performance for particularly structured grid problems that are used for solving PDE equations as well as image processing applications [2]–[4]. One of the difficulties of GPU programming for attaining good performance is that application developers need to have deep understanding of the host and device address spaces. CUDA being the most prominent GPU programming model unfortunately requires nontrivial programming effort. Ideally a programmer would like to write a single version of an application that will perform well on both homogenous multicore and heterogeneous multicore systems. Pragma-based programming models such as OpenMP or OpenACC promise to achieve this goal with code annotations [5]–[7]. These accelerator models are built on existing concepts (e.g. parallel region, parallel for loop) to provide a unified programming model for CPUs and GPUs. However, the achieved performance is typically suboptimal without aggressive performance tuning [8].

One of the main challenges of GPU programming is the management of data transfers between disjoint address spaces, which should be explicitly controlled by the programmer for best performance. Nvidia provides Unified Virtual Memory (UVM) as of CUDA 6.0, which gives the illusion of shared address space between the GPU and CPU. Although UVM enables simpler programming and lowers the barrier to CUDA programming, it provides far less performance. PCI Express (PCIe) link within the host has a considerably lower bandwidth than CPU or GPU memory bandwidths. NVLink is a high-bandwidth communication protocol between the CPU and the GPU, and between GPUs [9], which allows at least 5 times faster transfer speed than the current PCIe Gen3. While the NVLink technology improves the data transfer rate, the compute capability of GPUs continues to improve as well. The latest Pascal architecture reaches 5 Teraflops double precision performance for HPC workloads. As a result, in order not to lose the performance benefits of GPUs, application developers should hide data transfer latency.

To provide a uniform programming interface for CPUs and GPUs and to hide communication overhead, we introduce a tiling based programming model. We combine the concept of tiling, which is one of the techniques commonly used for optimizing applications for data locality [10], [11], with GPU programming in single model to overlap data transfers with execution of tiles on the device. In addition, decomposing data into tiles enables the programmer to run applications on GPU even though application data cannot entirely fit into the device memory. The implementation of the concepts leverages existing libraries and tools. The programming interface extends TiDA, a tiling library previously developed by Unat et al. [12] for multicore systems. We hide the data transfers between the GPU and CPU using CUDA streams and rely on efficient kernel code generation by OpenACC. In this paper, we demonstrate the effectiveness of the programming model implemented as a library on two different kernels and compare the performance with CUDA and OpenACC. One of these kernels represents the data transfer-intensive application, namely heat solver, where the data transfer cost dominates the execution time. The second kernel represents the compute-intensive kernel, where our library hides the transfer cost greatly. The performance results also show that the library achieves comparable performance with its counterparts.

This paper is organized as follows. Section II gives background on existing GPU execution models and compares their performance. Section III introduces the tiling based programming model. Section IV discusses the details of the library that implements the programming model. The application user interface is discussed in Section V followed by performance results in Section VI. We present related work in Section VII. Finally we conclude in Section VIII.

II. GPU Programming Models

In this section, we compare convenience and performance of CUDA, OpenACC and combination of both to find an execution model for our tiling based programming model. The GPU programming model requires the programmer to define functions called kernels that will be off-loaded to the GPU. The programmer also implements a host-side code that manages data transfers between the host and the device, sets up the kernel parameters, and launches kernels, which executes on the device. CUDA and OpenACC offer slightly different programming interfaces for these operations. CUDA is essentially C/C++ API with few extensions and gives the programmer greater control with detailed interface. OpenACC uses a pragma-based interface, which arguably provides a high level interface and high productivity to the programmer.

A. Kernels

In CUDA, programmer explicitly implements parallel kernels. Each kernel contains the code that is executed by a single thread. When invoking a kernel, the programmer specifies the number of threads in a block and number of blocks in a grid that concurrently executes the kernel. Tuning these parameters greatly affects the performance. The programmer can also tune the performance on the GPU by using on-chip memories.

In OpenACC, compiler generates the kernel code. Programmer starts a parallel section with a pragma as in OpenMP. Parallel sections are annotated with the *parallel* construct and inside a parallel section loops are annotated with the *loop* construct for GPU parallelization. If a loop is a tightly nested loop, programmer can compliment the *loop* construct with a *collapse* clause. The *collapse* clause takes an integer which indicates how many loops are tightly nested and can be parallelizable. For clarity, programmer can combine *parallel* with *loop* just before a tightly nested loop. There is also *kernels* construct in OpenACC to identify loop parallelism. While compiler relies on programmer's hints to map parallelism in a *parallel* construct, it automatically determines parallelism in a *kernels* construct by analyzing loops in the scope.

Depending on pragmas and their attributes, OpenACC compiler generates an optimized kernel. With the help of the pragma attributes, one can control kernel parameters. For example, *num_gangs*, *num_workers* and *vector_length* correspond to number of CUDA blocks in a grid, number of CUDA warps in a block and number of CUDA threads in a warp, respectively.

B. Memory Transfers

Main memories of the host and the device are physically disjoint and separated by the PCIe bus. Data that is shared by the host and the device must be allocated in both memories, and explicitly transferred between them by the application.

In a default memory transfer between the host and the device, CUDA copies data from pageable host memory to pinned host memory and then to device memory which is allocated with *cudaMalloc*. The copy between two host memories seems unnecessary but CUDA requires page-locking the data during the transfer. Even though CUDA runtime handles copies between two host memory locations, this operation lowers the transfer performance. To improve the performance of the transfers, CUDA provides an option, where it lets programmer lock (or pin) host memory and use this memory for data with the *cudaMallocHost* function. Moreover, NVIDIA introduced unified memory with CUDA 6.0 to alleviate some of the memory management complexity. With unified memory, programmer sees a single address space instead of two distinct address spaces and is not responsible for memory transfers. Using *cudaMallocManaged*, programmer allocates managed memory that can be accessed by the host and device with a single data pointer. CUDA runtime handles necessary data transfers internally where migration between two memories takes place on demand.

In OpenACC, memory management by the programmer is optional. Compiler automatically generates code for memory transfers before and after each generated kernel. However, this causes extremely low performance. Since compiler cannot have as much information as the programmer does about which data is needed and when on the GPU, memory management by the programmer becomes necessary for good performance. OpenACC provides memory management in two ways; with structured data lifetime and unstructured data lifetime. Programmer can create a scope with the *data* construct, which specifies the lifetime of the data in that scope. With the data scope, s/he can specify a memory copy or indicate that data is already in device or declare a pointer already pointing to device memory etc. For unstructured data lifetime, *enter data* and *exit data* directives are used. In this case, specified data is not limited to an enclosed scope. It can live on device from the beginning of the *enter data* until the *exit data*.

OpenACC also provides options to use pinned or managed memory instead of pageable memory with compiler flags for the Tesla GPU architectures. With the *-ta=tesla:pinned* flag, programmer can use pinned host memory. With the *-ta=tesla:managed* flag, programmer can use managed memory, which corresponds to unified memory in CUDA.

C. Performance of GPU Programming Models

We have implemented a simple stencil kernel which solves heat equation using CUDA only, OpenACC only and combination of both; then, compared their performance, convenience and functionality for two main GPU operations: data transfers and kernels. In the implementation of combined CUDA and OpenACC, we used CUDA for memory management and

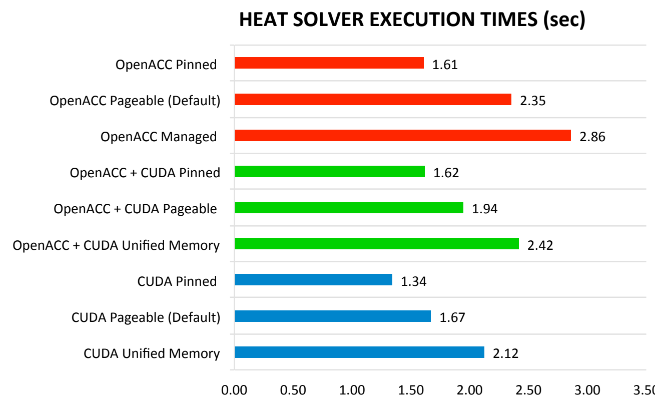

Fig. 1: Performance of different execution model implementations of heat equation. OpenACC + CUDA indicates implementations that rely on OpenACC for kernel code generation but uses CUDA for memory management.

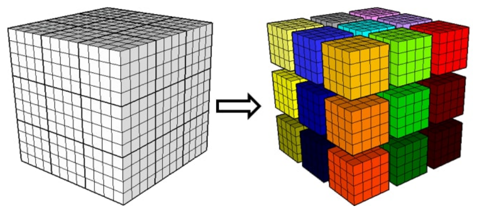

Fig. 2: CPU partitions data into regions.

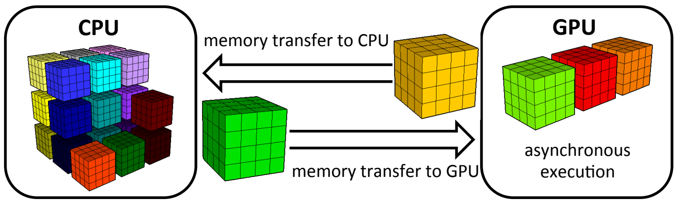

Fig. 3: While three regions are executed on GPU, two regions are transferred and rest of the regions can be executed on CPU.

OpenACC for kernel code generation. We also tested pageable, pinned and unified memory for each execution model. All implementations run with data of size 384^3 for 100 iterations on a NVIDIA Tesla K40m.

Figure 1 shows the running times of GPU implementations. Timing includes both data transfer and compute times. The CUDA-only pinned memory version has the best performance with the lowest running time. In all cases, pageable memory and virtual memory versions achieve lower performance than that of pinned as expected. OpenACC achieves lower performance than CUDA for the each memory management versions. When we manage the memory with CUDA pinned memory but rely on OpenACC for kernel code generation, the performance of OpenACC improves and gets much closer to that of CUDA. There are two reasons why CUDA still performs slightly better. Firstly, the pure CUDA implementation launches one kernel per time step to both calculate heat equation and update data boundaries. However, OpenACC implementations launch one kernel to calculate heat equation and multiple kernels to update data boundaries. Therefore, there is an overhead for each kernel launch. Second, we tune the grid and block geometry for the pure CUDA version; however, we let compiler decide the geometries for the OpenACC kernels. Note that none of the implementations use on-chip shared device memory.

Based on the results, we decide to use pinned host memory with CUDA for explicit memory management and data transfers, but leverage OpenACC for kernel code generation in our library implementation. This frees the users from implementing low level CUDA kernels or relying on in-house compiler. However, the user still benefits from higher performance that comes with CUDA pinned memory. Pinned host memory is also necessary for the library to increase concurrency with CUDA streams by overlapping operations such as memory transfers and kernel executions, which will be discussed shortly.

III. Programming with Tiles

Domain decomposition is commonly used for parallelism and data locality, but it can be also used for alleviating complicated management of distinct memories. In our execution model, we use domain decomposition and combine it with GPU acceleration to overlap computation and data transfers.

We manage the address spaces on both host and device memory and handle the memory transfers between them without programmer's involvement, which simplifies the GPU programming greatly. Basic execution unit and transfer unit in the execution model is the decomposed data, where we will refer them as *regions* through the paper. Data is decomposed into smaller physical memory spaces as shown in Figure 2. All regions are allocated on the device if there is enough device memory. If there is not enough storage, only a number of regions that can fit into the available device memory is allocated on the device while the rest resides on the host. This provides applications that do not fit into device memory take advantage of GPU acceleration.

Data transfers between CPU and GPU are costly because of the PCI-e bus bandwidth. Therefore, optimizing data transfers such as avoiding unnecessary ones and hiding their cost behind useful computation are essential for performance. To improve the performance of the tiling based execution model, we rely on two main techniques; caching and overlapping. We support a caching mechanism to prevent unnecessary data transfers between the two address spaces. We keep track of information about where each region is accessed last time and initiate a memory transfer on demand when a region is accessed at a different address space than it was last time. Unified virtual memory in CUDA also supports a caching mechanism; however, this mechanism offers lower performance.

There are two kinds of overlapping in the proposed execution model. The first one is overlapping memory transfers between host and device with execution as shown in Figure 3. While some of the regions are in GPU getting executed, some

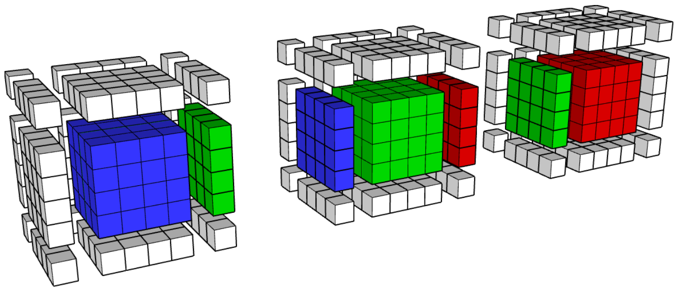
Fig. 4: Three regions (blue, green, red) with ghost cells. Colors on ghost cells indicate which ghost cell belongs to which region.

of them are in the interconnect getting transferred between CPU and GPU at the same time. The second one is overlapping computation in CPU with computation in GPU. Since data is partitioned, computations require neighbour data (ghost cells) from other regions. For this reason, regions are expanded to accommodate ghost cells. If data of a neighbor region changes, ghost cells in the related region should be updated. To prevent from unnecessary memory transfers, update of ghost cells of a region takes place in CPU or GPU depending on the location of the region at that moment. However, updating ghost cells involves branching and branch divergence is known to degrade performance on GPUs [13]. To prevent from branch divergence, we let CPU calculate the indices of ghost cells of a region and their correspondence on the source region. Then, CPU launches a kernel to update them. While GPU executes the kernel, host continues to calculate indices of other ghost cells of the region or ghost cells of another region. Figure 4 shows an example via three regions with the ghost cells. While blue ghost cells of the region in the middle is updated with a kernel on GPU, CPU calculates the indices of red ghost cells of the middle region and their correspondences on the region on its right.

IV. Implementation

We implemented the tiling based programming model that employs asynchronous GPU execution as a C++ library. The library leverages the TiDA API for programming with tiles [12], CUDA for memory management, OpenACC directives for kernel code generation, and finally CUDA streams with OpenACC interoperability for overlapping GPU operations. The library is built on top of abstractions introduced in TiDA with GPU extensions and we will refer to it as TiDA-acc in the rest of the manuscript.

A. *Tiling Abstractions*

The library employs tiling abstractions from TiDA, which originally has three abstractions: region, tile, and tile iterator. Regions are partitions of data and they are physically separated in memory. Tiles are logical partitions of regions. Unlike regions, tiles are not physically separated but they represent the iteration space belonging to a region. Tile iterator iterates over tiles (or regions) in an out-of-order fashion and manages parallelism. Tile and region sizes play an important role in performance. A programmer should pick a tile size to enable cache reuse and region size to have enough parallelism across different non-uniform memory access nodes. Since tile or region size changes does not require code modification, a programmer can easily tune these parameters. Alternatively, tools such as ExaSAT [14] can be leveraged to determine optimal sizes for working set and available cache.

In TiDA-acc, tiling abstractions have the same definitions for CPU execution. However, we add additional meanings to region and tile iterator abstractions for GPU execution. Regions are memory transfer units between host and device, and tile iterator is the engine that enables the GPU execution.

TiDA uses a data structure, `tileArray`, to help implement tiling abstractions. `tileArray` allocates memory for each region, does the partitioning of data, and keeps addresses of regions in a list. It is also responsible for updating ghost cells. In TiDA-acc, we enable `tileArray` to allocate pinned host memory with the *cudaMallocHost* function for regions because CUDA with pinned host memory offers the best performance as discussed in Section II. In addition to updating ghost cells on host, we also enable `tileArray` to initiate ghost cell update on device by calling a function of `TileAcc`, which will be discussed shortly.

B. `TileAcc`

`TileAcc` is the main data structure in the library and it is responsible for management of GPU-related operations. Next, we explain how `TileAcc` handles device memory management, streams, memory transfers, caching, kernel code generations, and finally ghost cell updates.

1) Memory Management: `TileAcc` allocates separate memories on device for each region. It calculates how many regions can fit into device memory by checking available memory with the *cudaMemGetInfo* function. It allocates device memories for that number of regions with *cudaMalloc*. Then, it keeps device memory pointers in a list by assigning their index as their ID. If there is enough memory on device for each region, this mapping is one-to-one. However, if there is not enough memory on device, some of the regions share the same device memory pointer, allowing applications that cannot fit into device memory still run on device. The mapping occurs between ID of a region assigned by `tileArray` and ID of a device memory pointer, and specifies which regions share the same device memory pointers. Each device memory pointer has also a CUDA stream assigned to it.

2) Streams: `TileAcc` uses streams to overlap communication and computation on a GPU. CUDA streams are sequences of device operations. Operations in a single stream is serially executed one after another by the device. By default, device operations are associated with a default CUDA stream. However, programmer can associate different operations to different streams to overlap these operations. OpenACC also offers the same feature to increase concurrency. CUDA streams are referred as activity queues in OpenACC. Programmer uses *async(int async-value)* clause to queue a device operation to the queue with *async-value* identifier.

Combining CUDA and OpenACC in the same program is possible. For this purpose, OpenACC provides runtime functions. For example, *acc_get_cuda_stream(async-value)* function returns a CUDA stream associated with OpenACC activity queue, so that programmer can issue desired CUDA and OpenACC operations to the same stream or activity queue. `TileAcc` uses this interoperability and creates streams with the *acc_get_cuda_stream(async-value)* function and keeps them in a stream list. Each stream is assigned to a device memory pointer based on their index in the stream list.

3) Memory Transfers: `TileAcc` handles memory transfers with CUDA and uses regions as transfer units. It queues memory transfers of the regions to their corresponding streams by calling the *cudaMemcpyAsync* function. All the memory transfers are asynchronous. Moreover, `TileAcc` does not need synchronization in transfers from host to device because streams preserve the execution order both for data transfers operations and kernels launched on a region. However, it needs synchronization for transfers from device to host because programmer may access data right after requesting it on host. Therefore, instead of returning host data pointer after queuing the transfer, `TileAcc` waits for transfer to be completed by using the *cudaStreamSynchronize* function.

4) Caching: `TileAcc` keeps track of where the most up-to-date data is, on the host or on the device, and eliminates unnecessary data transfers. It creates a list with size equal to the number of allocated device memory pointers. Each index in the list corresponds to a device memory pointer and value at each index corresponds to ID of the region which uses associated device memory pointer.

At the beginning, all the values in cache list is -1, indicating no region's data is in device. When programmer requests a region in a GPU enabled iteration, `TileAcc` first checks if the region is in device or not by looking at the cache list. Then, if data of the region is on device, i.e. if the value in the cache list is equal to region's ID, it returns device data pointer of the region directly. However, if the region is not on device, there are two possibilities. First possibility is that assigned device memory pointer is not in use, i.e. the value in the cache list is equal to -1. In this case, `TileAcc` queues memory transfer of the region and changes the value in cache list to its region ID. Second possibility is the region might be sharing the same device memory pointer with another region and that pointer might be currently in use in case of not available device memory for all regions. If this is the case, value at corresponding index in cache list equal to neither ID of requested region, nor -1. `TileAcc` queues memory transfers of the region that is currently using the assigned device memory pointer from device to host in its stream before it queues transfers of the requested region. After queuing the transfer of the requested region, it changes the cache value to the region ID. If programmer requests a region on CPU, i.e. GPU disabled iteration, `TileAcc` performs a check on cache list to see if the region is on device. If the region is not on device, it directly returns the host region pointer. If the region is on device, it queues a transfer to host in the stream of the requested region.

5) Kernels: `TileAcc` generates kernels with the help of OpenACC directives encapsulated in a static *compute* method that user calls. The method abstracts directives and creates a uniform interface which does not require any change in the source code for both CPU and GPU executions. It takes tile(s) that will be computed and a C++ lambda function. It also takes optional arguments, which we explain in more detail in Section V. The lambda function contains the computation that is going to be applied to tiles.

In the *compute* method, there is a nested for-loop in which lambda function is called. In the library implementation, we annotate the nested loop with an OpenACC directive, *acc parallel loop collapse(...) deviceptr(...) async(...)*. The $collapse(...)$ clause is used to specify the number of tightly nested for-loops. The $deviceptr(...)$ clause is used to indicate data pointers as device data pointers. The $async(...)$ clause is used to issue the kernel to the stream of the region. The OpenACC compiler generates kernel code in compile time based on these directives. Then, each kernel is issued to necessary stream at runtime. After issuing kernel, no synchronization is needed.

6) Ghost Cell Update: TiDA-acc provides a mechanism to update ghost cells on device. Depending on the last location of a region's data, i.e. on CPU or GPU, `tileArray` updates ghost cells by itself on CPU or initiates the mechanism provided by `TileAcc`. `TileAcc` utilizes both CPU and GPU when updating ghost cells. While it calculates source and destination indices of a set of ghost cells in a region on the host, it also executes a kernel, which updates different set of ghost cells in the same region or ghost cells in different regions on the GPU. Before starting to calculate indices and update ghost cells, it synchronizes all the executions in all streams of GPU with the $acc\ wait$ pragma. Then, for each region, it computes the source regions of ghost cells, calculates indices and launches kernels for update. It queues kernels to the stream of the region whose ghost cells are updated with the async clause of OpenACC. Since, streams are queued operations and each region has its own stream, the execution order is preserved. Therefore, we do not need a synchronization point after queuing kernels for ghost cell update.

V. API

In this section, we introduce the user interface of the library. The main goal of the accelerated tiling library is that programmer does not need to deal with management of address spaces, memory transfers, kernel implementations, or compiler directives. The library handles all these operations on the behalf of the programmer. Programmer only needs to indicate the intention of GPU acceleration in a tile iterator and use a library method, $compute$.

The simple code below demonstrates the user interface. Programmer defines a `tileArray` with a region size that will partition the data into regions and then defines a tile iterator, *tIter*, to traverse over tiles in a loop. Tile iterator optionally takes logical tile size for loop iteration space partitioning. In the initialization part of the for-loop, programmer resets the iterator with an argument, which enables GPU execution for the loop. After enabling GPU execution, the *compute* function will launch a kernel for a tile. The loop runs until the iterator consumes all the tiles. If a programmer uses a tile size which is smaller than size of a region, there will be multiple tiles for a region to iterate. On CPU, using multiple tiles for a region is desirable to enable cache reuse. On GPU, however, using multiple tiles may degrade performance because there will be multiple kernel launches for a region. As a result, it is recommend to use tile size equal to region size when GPU execution is enabled.

```
1  TileArray ta = TileArray(data_size,
2                             region_size, ghost_size);
3   . . .
4  TileIter tIter = TileIter(&ta, tile_size);
5  for(tIter.reset(GPU=true); tIter.isValid(); tIter.next())
6  {
7    Tile tile = tIter.getTile(ta);
8    TiDA::compute(tile, [&](double* data, int depth,
9                   int height, int width, int index) {
10   // compute
11   data[index] = ...
12   });
13 }
```

Programmer enables GPU execution for the loop in line 5 and requests a tile from the iterator in line 7. Instead of using nested for-loops to iterate over the data in a tile, in line 8 programmer uses the *compute* method provided by TiDA-acc. The method takes two arguments; a tile to iterate over and a lambda function, in which programmer implements the desired computation. The lambda function expects a list of arguments as well. In the code example, parameters are a pointer for the tile's data, dimensions of the data and an index. If computation involves multiple tiles as inputs, then the *compute* method takes these tiles and a lambda function as arguments. The parameter list of lambda function should also have the same number of data pointers to the tiles' data. The first double pointer argument of lambda function corresponds to data pointer of the first tile in parameter list of the *compute* method and so on as shown in the code below.

```
for(tIter.reset(GPU=true); tIter.isValid(); tIter.next())
{
 Tile tile_a = tIter.getTile(tileArray_a);
 Tile tile_b = tIter.getTile(tileArray_b);
 TiDA::compute(tile_a, tile_b, [&](double* data_a, double
     * data_b, int depth, int height, int width, int
     index) {
  // compute
  data_a[index] = data_b[index + width] ...
 });
}
```

Programmer may also need to iterate over a specific range in a tile. In this case, the *compute* method takes two dimensions. First dimension indicates lower bounds and second dimension indicates upper bounds of the desired iteration space. The code example below demonstrates this case.

```
for(tIter.reset(GPU=true); tIter.isValid(); tIter.next())
{
 Tile tile = tIter.getTile(tileArray);
 Dimension lb = Dimension(1,1,1); //lower bound
 Dimension ub = Dimension(16,16,16); //upper bound
 TiDA::compute(tile, lb, ub, [&](double* data, int depth,
      int height, int width, int index) {
  // compute
  data[index] = ...
 });
}
```

A. Limitation

Library uses the *compute* method to hide the necessary kernel generation code and OpenACC directives. Programmer calls this function to iterate over data and provides a lambda function that contains computation as an argument. The disadvantage of iterating over data with the *compute* method is that it prevents usage of imperfectly nested loops. Instead of the *compute* method, a method with a similar structure to a regular for-loop would be ideal such as this one:

```
custom_for(int start, int end, int increment, LambdaFunc
     function);
```

In this case, we could be able to use a custom for-loop and support imperfectly nested loops as follows:

```
tida_for(0, data_size-1, 1, [&](int i){
      // compute
      data[i] = ...
});
```

However, due to the current limitation of OpenACC support for lambda functions, we chose to implement the kernel launches through the *compute* method. Assuming the loop above is in a GPU-enabled traversal, then *data* would be a device pointer. Under OpenACC unless *data* is an argument to the lambda function, it cannot be declared as a device pointer. No matter where and how the declaration is, declaring the pointer as a device pointer does not generate the correct kernel code and gives illegal address at runtime.

In OpenACC, when a function is annotated with a *routine* directive, compiler generates a device version of that function, which brings the advantage of having function calls within a kernel. If a pointer is declared as a device pointer in a scope, the declaration also applies to device versions of the functions called from that scope. Even though OpenACC generates a device version of a lambda function implicitly, the device pointer declarations do not apply to them. As a solution, we declare a data pointer as a device pointer in the library and give it as a parameter to lambda function. Therefore, the programmer needs to structure the lambda function in a way that it takes the data pointer as a parameter. As OpenACC improves its support for C++ lambda functions, we will reconsider this design decision.

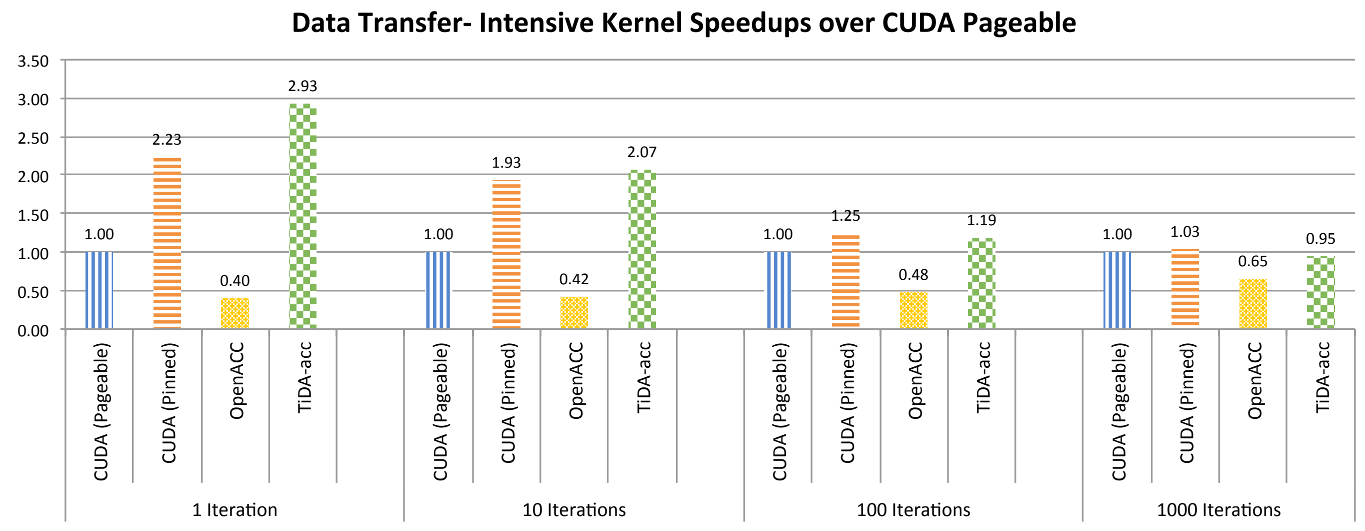

Fig. 5: Speedups of transfer-intensive kernel in different execution models.

VI. Results

We evaluate performance of TiDA-acc on two kernels. In one of the kernels, memory transfers take more time than computation and we refer to it as data transfer-intensive application. In the other kernel, computation takes more time than memory transfers and we refer to it as compute-intensive.

Our setup has Intel Xeon E5-2695 v2 processors as the host and a NVIDIA Tesla K40m as the GPU. We use PGI compiler version 17.1.0 for OpenACC and NVCC compiler version 7.5.17.

A. *Data Transfer-Intensive Kernel*

We chose the heat solver as the data transfer-intensive application. Normally the running time of this application is dominated by the data transfer cost unless the programmer runs the simulation for a large number of time steps to amortize the data transfer cost assuming that all the application data fits into the device memory [15]. The main computation uses its 6 nearest neighbors to solve the heat equation for each cell at each time step, i.e iteration. Following code shows the main computation.

```
data_next[cell] = data[cell] - alpha * (data[front_cell]
  + data[back_cell] + data[top_cell] + data[bottom_cell]
  + data[left_cell] + data[right_cell]);
```

Figure 5 shows the speedup over the CUDA pageable memory for CUDA pinned memory, OpenACC with pageable memory and TiDA-acc implementations of the heat solver for different number of iterations. We could not include the OpenACC with pinned host memory implementation because it fails to generate correct code and complains about insufficient device memory. We used data with size 512^3 and ran the application for $1, 10, 100$ and 1000 iterations. The execution times include both memory transfer time and computation time. Note that CUDA and CUDA pinned memory implementations are not optimized, i.e. not using shared memory or texture memory. However, block and grid geometry of kernels are tuned in these executions. For the TiDA-acc implementation, we used 16 regions which gave the best performance.

The results show that TiDA-acc performs reasonably well especially when the the number of time steps is low. The reason for this is that both CUDA and OpenACC versions are penalized by the data transfer cost and the computation done on the GPU cannot ammortize this overhead. Since TiDA-acc breaks the data into smaller tiles, it can hide the transfer latency with the computation on the tiles. As the number of iterations increases, both CUDA versions achieves similar performance as TiDA-acc. Since we use OpenACC for kernel code generation, the advantage of TiDA-acc is clearly seen as OpenACC without asynchronous data transfer support has the lowest performance of all.

B. *Compute-Intensive Kernel*

We adopted the compute-intensive kernel, which is originally developed by NVIDIA as a benchmark to show the potential overlap of memory transfers and kernel executions [16]. The kernel adds sum of squares of sine and cosine of each cell to itself. Since the original kernel is designed to obtain equal memory transfer time to kernel execution time for an older generation of GPUs, we added a loop to iterate over the computation to increase its computation time to adjust it on our target device. There is also an outer loop iteration outside of the kernel that resembles the time step iterations in the heat solver. Following is the code that a kernel executes at each time step.

```
double s, c;
for(double i=0; i<kernel_iteration; i++){
 s = sin(data[index]);
 c = cos(data[index]);
 data[index] = data[index] + sqrt(s*s+c*c);
}
```

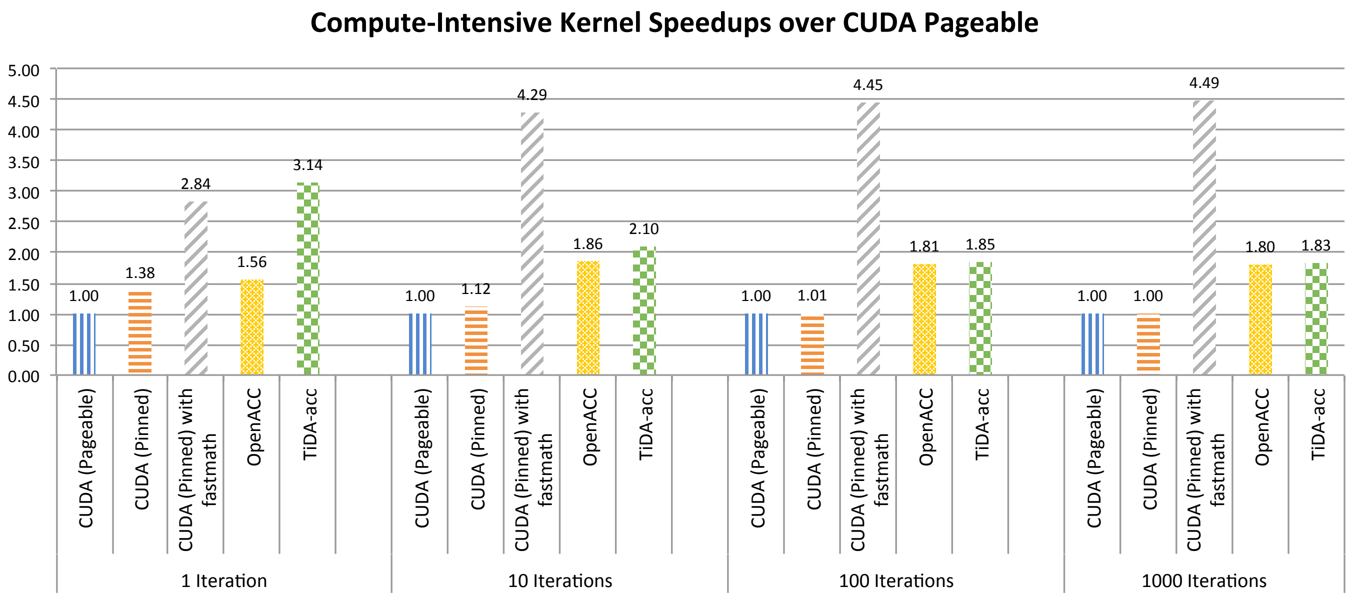

Fig. 6: Speedups of compute-intensive kernel in different execution models.

Figure 6 shows the execution times of implementations of CUDA, CUDA pinned memory, CUDA pinned memory with fast math, OpenACC with pageable memory and TiDA-acc. The execution times include both memory transfer time and computation time for data size of 512^3. We use standard *math.h* library in the source code of OpenACC and TiDA-acc versions. Since the PGI compiler handles kernel code generation of math functions for these versions, these versions run faster than the CUDA variant. For the sake of fairness to CUDA, we also included a version of CUDA that turns on *use-fast-math* flag for faster execution with lower precision. Results in Figure 6 shows that TiDA-acc performs reasonably well as it does not introduce any overhead. The performance of OpenACC is also comparable because this kernel does not require ghost cell exchange as the data transfer-intensive kernel.

C. Limited Memory Case

One of the important contributions of TiDA-acc is that it can run an application on GPU even if there is not enough GPU memory to hold all the application data. For example, Figure 7 illustrates TiDA-acc execution with limited memory on GPU. There are two streams: s1 and s2. *D2H* and *H2D* indicate memory transfer from device to host and from host to device. *C:R#* is the computation on region ID#. The figure shows that data transfers are fully overlapped with computation on GPU, which allows execution on limited memory without any performance loss.

To demonstrate this, we run compute-intensive kernel by limiting GPU memory to hold only two regions. For the data size of 512^3 and 1000 time steps, Figure 8 compares the performance of TiDA-acc with no limitation on GPU memory (shown as TiDA-acc) and TiDA-acc with limited GPU memory. In the limited memory case, CUDA cannot run the application on GPU, but the library handles such situation and shows almost the same performance with the available memory case. Moreover, we also show a third version of TiDA-acc (TiDA-acc with 1 region), which does not divide the data into regions but creates a single big region for all the application data as in CUDA. For the one region case, the library does not introduce any overhead as it performs similar to TiDA-acc, where the application data is divided into multiple regions.

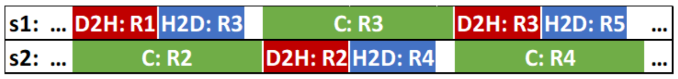

Fig. 7: Illustration of executions for multiple regions with a complete overlap of data transfers and computation.

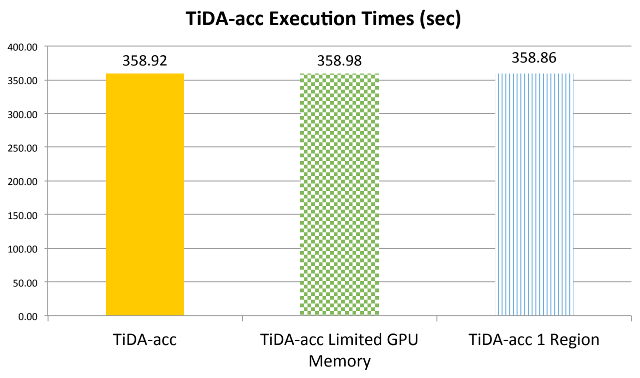

Fig. 8: Comparing TiDA-acc with one region and limited memory cases in compute-intensive kernel

VII. Related Work

There have been a large number of research contributions from various groups to facilitate programming on GPUs since GPUs started being used as a general purpose processors. While some of these works [17]–[20] rely on CUDA, some [7], [21] are based on OpenCL for portability. The approaches that rely on CUDA either require programmer to implement the kernels explicitly or use in-house compiler to generate CUDA code. Our approach utilizes CUDA in data transfers for better performance but leverages an OpenACC compiler to generate efficient kernel code. XACC [22], a directive based language that targets multi-GPUs also leverages OpenACC for kernel code generation however XACC does not provide asynchronous data transfers to overlap data movement with computation.

With the inclusion of anonymous function (lambda function) support in C++11, many groups [20], [23]–[25] have proposed using lambda functions to hide some of the complexities of GPU programming. Kokkos [23] offers performance portability for different parallel programming models with C++ templates and lambda functions. Kokkos abstracts kernel code generation with CUDA lambda functions introduced in CUDA 7.5. However, it does not completely hide distinct memory management, i.e. host and device memories, from the programmer. Instead, it offers a high level memory management with C++ templates. In Kokkos, programmer uses *views* to manage memory and should indicate memory as a device memory. In Thrust [25], programmer uses *thrust::host_vector* and *thrust::device_vector* for memory management. RAJA [20] also supports GPU programming with C++ templates and lambda functions although programmer needs to map the data between host and device. For example, in [26], programmers use RAJA with OpenMP 4.1 and map data in the constructor of a data structure with unstructured data mapping support of OpenACC. SYCL [21] also abstracts kernel code generation with lambda functions, but requires data to be placed in device buffers, which is not a complete abstraction. C++ AMP [27] uses lambda for kernel code generation, and implicitly transfers memory. It requires constructing *array_view* objects for arrays that will be used for GPU execution. PACXX [28] abstracts kernel code generation and memory management without additional structures by extending *std::array* and *std::vector*, but this limits the data structures that programmer can use. Moreover, none of these programming models provide overlapping of memory transfers and kernel execution. Hiding data movement overhead is at the core of our programming model.

As a recent study, CuMAS [29] offers automatic overlapping of data transfers and kernel executions, but it focuses on scheduling multiple CUDA applications, rather than scheduling of a single application's data transfers. dCUDA [30] is a runtime system that overlaps computation with inter-node communication on a multi-GPU environment but it relies on the programmer to implement the CUDA kernels. Daino [31], a compiler-based framework for executing Adaptive Mesh Refinement (AMR) applications on GPUs, requires user directives but its runtime hides many details of data movement.

Our proposed technique is based on physically and then logically decomposed data, where tiles constitute the basic unit of data transfer and execution. HTA (Hierarchically Tiled Arrays) [32] describes hierarchy of tiles and uses overloaded array operations to represent computation and communication. We benefitted the design principles of HTA with a performance focus and extend the TiDA [12] library for GPUs. TiDA has been adopted by the BoxLib AMR Library [33] for tiling on multicore systems. Others [34] used tiling for synchronization avoidance and parallelism on GPUs with polyhedral compiler, rather than for data decomposition.

VIII. Conclusion

In this paper, we introduced a tiling based programming model for GPUs that overlaps the computation and communication between the host and the device. The core concept in the programming model is that it divides the data and computation into tiles and treats tiles as the main execution and transfer units. This enables the programmer to both hide the communication overhead and execute applications whose data cannot fit into the device memory. The programming model is implemented as a library which leverages existing tools to achieve good performance and programmer's productivity. The library uses the TiDA tiling library for its API, utilizes C++ lambda function to encapsulate OpenACC directives from the programmer and relies on CUDA streams to handle asynchronous data transfers between the host and the device. Our performance results on two test cases show that the achieved performance is reasonable and the library greatly hides the communication overhead. Moreover, the library can handle cases where the device memory is not sufficient to hold the entire application data by staging the data as tiles.

Acknowledgments

Authors from Koç University are supported by the Turkish Science and Technology Research Centre Grant No: 215E185. Dr. Unat is supported by the Marie Sklodowska Curie Reintegration Grant 655965 by the European Commission. Authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231.

References

[1] "http://www.top500.org/."

[2] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 79–84, ACM, 2009.

[3] H. S. Kim, D. Unat, S. B. Baden, and J. P. Schulze, "A new approach to interactive viewpoint selection for volume data sets," *Information Visualization*, vol. 12, no. 3-4, pp. 240–256, 2013.

[4] J. Zhou, D. Unat, D. J. Choi, C. C. Guest, and Y. Cui, "Hands-on performance tuning of 3d finite difference earthquake simulation on gpu fermi chipset," *Procedia Computer Science*, vol. 9, pp. 976 – 985, 2012.

[5] S. Lee and R. 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, pp. 1–11, IEEE Computer Society, 2010.

[6] M. Wolfe, "Implementing the PGI Accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pp. 43–50, 2010.

[7] J. Kim, Y.-J. Lee, J. Park, and J. Lee, "Translating openmp device constructs to opencl using unnecessary data transfer elimination," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, (Piscataway, NJ, USA), pp. 51:1–51:12, IEEE Press, 2016.

[8] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 136–143, May 2013.

[9] NVDIA, "NVDIA NVLINK High-Speed Interconnect: Application Performance," tech. rep., 2014.

[10] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3D scientific computations," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, SC '00, IEEE Computer Society, 2000.

[11] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericas, "Trends in data locality abstractions for hpc systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2017.

[12] D. Unat, T. Nguyen, W. Zhang, M. N. Farooqi, B. Bastem, G. Michelogiannakis, A. Almgren, and J. Shalf, "TiDA: High-Level Programming Abstractions for Data Locality Management," in *Proceedings of 31st International Conference on High Performance Computing, ISC High Performance 2016*, pp. 116–135, Springer International Publishing, 2016.

[13] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in gpu programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, (New York, NY, USA), pp. 3:1–3:8, ACM, 2011.

[14] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf, "Exasat: An exascale co-design tool for performance modeling," *The International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 209–232, 2015.

[15] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 4:1–4:12, IEEE Press, 2008.

[16] M. Harris, "How to overlap data transfers in cuda c/c++." https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/, December 2012.

[17] D. Unat, X. Cai, and S. B. Baden, "Mint: Realizing cuda performance in 3d stencil methods with annotated c," in *Proceedings of the International Conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 214–224, ACM, 2011.

[18] S. Lee and R. 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, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[19] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance cuda code," *ACM Trans. Archit. Code Optim.*, vol. 9, pp. 31:1–31:25, Jan. 2013.

[20] J. K. Rich Hornung, "The RAJA Portability Layer: Overview and Status," in *Lawrence Livermore National Laboratory, Technical Report LLNL-TR-661403*, ACM, 2014.

[21] R. Keryell, R. Reyes, and L. Howes, "Khronos sycl for opencl: a tutorial," in *Proceedings of the 3rd International Workshop on OpenCL*, p. 24, ACM, 2015.

[22] M. Nakao, H. Murai, T. Shimosaka, A. Tabuchi, T. Hanawa, Y. Kodama, T. Bokut, and M. Sato, "Xcalableacc: Extension of xcalablemp pgas language using openacc for accelerator clusters," in *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, (Piscataway, NJ, USA), pp. 27–36, IEEE Press, 2014.

[23] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *2013 Extreme Scaling Workshop (xsw 2013)*, pp. 18–24, Aug 2013.

[24] M. Bianco and B. Cumming, "A generic strategy for multi-stage stencils," in *Euro-Par 2014 Parallel Processing* (F. Silva, I. Dutra, and V. Santos Costa, eds.), vol. 8632 of *Lecture Notes in Computer Science*, pp. 584–595, Springer International Publishing, 2014.

[25] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," *GPU computing gems Jade edition*, vol. 2, pp. 359–371, 2011.

[26] A. C. Jacob, S. F. Antao, H. Sung, A. E. Eichenberger, C. Bertolli, G.-T. Bercea, T. Chen, Z. Sura, G. Rokos, and K. OBrien, "Towards performance portable gpu programming with raja," *Workshop on Portability Among HPC Architectures for Scientific Applications*, 2015.

[27] K. Gregory and A. Miller, "C++ amp: accelerated massive parallelism with microsoft visual c++," 2014.

[28] M. Haidl, B. Hagedorn, and S. Gorlatch, "Programming gpus with C++14 and just-in-time compilation," in *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK*, pp. 247–256, 2015.

[29] M. E. Belviranli, F. Khorasani, L. N. Bhuyan, and R. Gupta, "Cumas: Data transfer aware multi-application scheduling for shared gpus," in *Proceedings of the 2016 International Conference on Supercomputing, ICS 2016, Istanbul, Turkey, June 1-3, 2016*, pp. 31:1–31:12, 2016.

[30] T. Gysi, J. Bär, and T. Hoefler, "dcuda: hardware supported overlap of computation and communication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pp. 52:1–52:12, 2016.

[31] M. Wahib, N. Maruyama, and T. Aoki, "Daino: A high-level framework for parallel and efficient amr on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, (Piscataway, NJ, USA), pp. 53:1–53:12, IEEE Press, 2016.

[32] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, (New York, NY, USA), pp. 48–57, ACM, 2006.

[33] W. Zhang, A. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, "Boxlib with tiling: An adaptive mesh refinement software framework," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S156–S172, 2016.

[34] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for gpus: Automatic parallelization using trapezoidal tiles," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, (New York, NY, USA), pp. 24–31, ACM, 2013.