IEEE IPDPS, Orlando, Florida, USA, May, 2017
2017 IEEE International Parallel and Distributed Processing Symposium

Directive-Based Partitioning and Pipelining for Graphics Processing Units

Xuewen Cui
Virginia Tech
Blacksburg, VA 24060
xuewenc@vt.edu

Thomas R. W. Scogland
Lawrence Livermore National Laboratory
Livermore, CA 94550 USA
scogland1@llnl.gov

Bronis R. de Supinski
Lawrence Livermore National Laboratory
Livermore, CA 94550 USA
bronis@llnl.gov

Wu-chun Feng
Virginia Tech
Blacksburg, VA 24060
feng@cs.vt.edu

***Abstract*—The community needs simpler mechanisms to access the performance available in accelerators, such as GPUs, FPGAs, and APUs, due to their increasing use in state-of-the-art supercomputers. Programming models like CUDA, OpenMP, OpenACC and OpenCL can efficiently offload compute-intensive workloads to these devices. By default these models naively offload computation without overlapping it with communication (copying data to or from the device). Achieving performance can require extensive refactoring and hand-tuning to apply optimizations such as pipelining. Further, users must manually partition the dataset whenever its size is larger than device memory, which can be especially difficult when the device memory size is not exposed to the user.**

We propose a directive-based partitioning and pipelining extension for accelerators appropriate for either OpenMP or OpenACC. Its interface supports overlap of data transfers and kernel computation without explicit user splitting of data. It can map data to a pre-allocated device buffer and automate memory-constrained array indexing and sub-task scheduling. We evaluate a prototype implementation with four different applications. The experimental results show that our approach can reduce memory usage by 52% to 97% while delivering a $1.41\times$ to $1.65\times$ speedup over the naive offload model.

I. Introduction

Systems with accelerators, particularly GPUs, are becoming prominent on the Top500 [1]. Many programming models support these systems, but rather than grapple with unfamiliar programming models scientists often prefer to keep their existing verified C, C++ or Fortran code. OpenMP since version 4.0 [2], [3] and OpenACC [4] allow the straightforward adoption of that existing code. Without extensively rewriting their code, users can add directives to offload their computation to accelerators.

OpenMP and OpenACC have similar offload mechanisms. Users annotate data with mapping or copying directives to ensure that the accelerator can access the data. They then launch computation on the accelerator and ensure that the results are available on the host when needed. If the accelerator cannot access host memory directly or if the use of memory associated with the accelerator can improve performance, then the data is copied to device memory, which can take significant execution time when performed synchronously (i.e., the naive offload model). Thus, programmers often work hard to overlap the transfers with computation. Further, the data may not fit in device memory because scientific applications frequently use huge data arrays or matrices. In this case, the user must manually split the data and the associated computation, which can involve significant code changes since the user must partition large data arrays on the host and separately pass each array pointer.

We extend OpenMP to partition data and to overlap transfers with computation through pipelining automatically. Our extensions allow data to be mapped into a small buffer to reduce memory usage and offer a simple interface to pipeline a parallel loop with an index handler and a kernel scheduler. Our experimental results show that our techniques can reduce memory usage and improve performance significantly.

This paper makes the following contributions:

- A comprehensive study that identifies limitations of current programming extensions for GPU devices;
- A new directive-based pipelined extension for OpenMP that automates the overlap of data transfers and kernel computation and reduces GPU memory consumption;
- A prototype implementation of our approach;
- A detailed evaluation of our approach for four applications on two diverse GPU architectures.

Our results demonstrates that our approach can provide a $1.41\times$ to $1.65\times$ speedup while reducing memory usage 52% to 97% over the naive offload model.

The rest of the paper is organized as follows: In Section II, we introduce GPU programming models and discuss the pipelining of data transfers. Section III describes our directive-based pipelining extension, which improves programmability, performance, and memory management. Section IV presents our prototype implementation and the use of our approach in four applications. In Section V, we compare our prototype with current programming models.

II. Background

Supercomputers increasingly have accelerators, such as GPUs, FPGAs, APUs, and co-processors like the Intel Xeon Phi to increase their performance per watt and performance per dollar. Programming these accelerators requires the use of alternate programming models or language extensions such as CUDA, OpenMP, OpenACC, and OpenCL.

1530-2075/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPS.2017.96

OpenMP is a directive-based extension for Fortran, C and C++ that is best known for providing portable multithreading on shared-memory multicore systems. A loop can be parallelized by inserting an OpenMP directive, and further custom parallelism can be achieved with a combination of directives and runtime API calls. Since OpenMP 4.0, OpenMP has included device constructs that *target* offload to devices with potentially distinct memory spaces. This OpenMP support for accelerators is still relatively nascent, offering opportunities for improvement [5].

CUDA, a parallel computing framework from NVIDIA, targets NVIDIA GPUs. It is one of the most widely used programming models for GPUs despite its lack of portability. It provides a powerful, flexible programing interface with low-level GPU control. GPU threads are grouped as a grid of thread blocks, which are mapped to GPU streaming multiprocessors. CUDA often requires the programmer to re-factor their code significantly [6].

OpenCL is a standard, low-level model that the Khronos group maintains. OpenCL implementations offer portability across GPUs, multicore CPUs, DSPs, co-processors, and FPGAs. However, OpenCL's complex, very low-level API often requires significantly more code than even CUDA.

OpenACC directives define compute and data regions in C, C++, and Fortran programs. As with OpenMP, the programmer must identify the region to offload with directives. An unofficial offshoot of OpenMP, OpenACC provides little support for host devices or CPU-like accelerators, limiting them to the same constructs as are applied to GPUs. Nonetheless, current implementations are more mature than existing support for OpenMP 4.X. Thus, several studies have compared its directive-based approach to CUDA in terms of performance, portability, and programmability [7], [8], [9].

III. Design

Our extension automates the implementation of overlapped computation and data transfers and of computation on data arrays that are too large for device memory. Users neither need to re-factor their code nor to break down work manually. We overcome three main challenges. First, we must correctly sequence concurrent data transfers and kernel computation. Second, we must partition the computation and data so all necessary inputs and outputs for a partition fit in memory on the target device. Third, we must modify array indices to reflect the partitioned accesses.

We address these challenges by dividing the loop into several smaller chunks, which we launch with their required data transfers on different GPU streams. When the transfer of the first chunk finishes, its kernel begins execution. Each chunk's transfers are enqueued separately, and thus may run in parallel. We control the chunk size in the runtime system automatically to avoid exceeding available memory. Chunk size can be explicitly determined with the `schedule()` clause if desired. Future work will integrate a performance model into an auto-tuning scheduler for better performance.

```
#pragma omp target\
   pipeline(schedule_kind[chunk_size,num_stream])\
   pipeline_map(map_type:array_split_list)\
   pipeline_mem_limit(mem_size)
```

pipeline() inputs

<schedule_kind>	scheduler to use for this region(static, adaptive).
<chunk_size>	sub-task chunk size.
<num_stream>	stream number to launch on GPU.

pipeline_map() and pipeline_mem_limit() inputs

<map_type>	to/from/tofrom for input/ output / input & output arrays.
<array_split_list>	array declaration
<mem_size>	maximum memory usage

array_split_list structure

<var>	variable(array) to copy
[split_iter:size]	split_iter: split start offset, size: split range
[0:m]	other non-related dimensions

Figure 1: Our proposed pipeline extension for OpenMP

Our framework calculates dependencies of the current chunk and removes the data that only previous chunks require. By mapping the data array to a small pre-allocated device buffer, we copy new data arrays into the location of this stale data inside the buffer. Thus, by mapping the segment of data for a chunk into a small buffer, we significantly reduce the memory requirement of many kernels.

Figure 1 presents the clauses that our extension adds. The `pipeline_map` clause extends the semantics of the `map` clause, which makes all data available at the beginning and/or at the end of the region. Our `pipeline_map` clause splits the data updates and subsequent loop computation into multiple subtasks. As with the `map` clause, the `map_type` specifies the data transfer direction.

The `array_split_list` is a new parameter that defines how to split the arrays. The format of this parameter is `<var>[split_iter:size][0:m]`. The `<var>` is the variable or base pointer of an array. The `[split_iter:size]` parameter identifies the dimension to split while `split_iter` is a function of the loop variable of the subsequent loop. The function defines the split starting offset in that dimension while the `size` defines the range. The split currently can be performed in one or two dimensions since our runtime system supports 1D and 2D memory copies. This `size` parameter helps us determine the array offset. We use different internal APIs for data movement based on the subsequent loop, which we discuss later. The `[0:m]` parameter defines the other dimensions, which do not impact the split. The parameter helps us determine the array size. The `pipeline()` clause specifies which schedule to use; currently we only support *static*, but future work will support adaptive schedules.

The `chunk_size` is the number of indices in the subsequent loop that we handle in each device buffer (potentially fewer in the last chunk). The `num_stream` parameter determines the number of GPU streams used. This parameters determines the number of chunks that we launch asynchronously. We choose these two parameters as the key components of our framework not only because they

```
#pragma omp target \
  pipeline(static[1,3])\
  pipeline_map(to:A0[k-1:3][0:ny-1][0:nx-1])\
  pipeline_map(from:Anext[k:1][0:ny-1][0:nx-1])\
  pipeline_mem_limit(MB_256)
for(k=1;k<nz-1;k++) {
#pragma omp target teams distribute parallel for
  for(i=1;i<nx-1;i++) {
    for(j=1;j<ny-1;j++) {
      Anext[Index3D (i,     j,     k)] =
        (A0[Index3D (i,     j,     k + 1)] +
         A0[Index3D (i,     j,     k - 1)] +
         A0[Index3D (i,     j + 1, k)] +
         A0[Index3D (i,     j - 1, k)] +
         A0[Index3D (i + 1, j,     k)] +
         A0[Index3D (i - 1, j,     k)])*c1
       - A0[Index3D (i,     j,     k)]*c0;
    } }
}
```

Figure 2: A stencil benchmark example

provide information to improve the data transfer between host and device, but also because they can significantly affect performance and memory use.

We can also limit memory usage with the `pipeline_mem_limit()` clause. The `num_stream` and `chunk_size` parameters determine the size of the device buffer, which we tune before we allocate the buffer to fit total memory usage within available size. The other target clauses, for example, `device` or `private`, work as previously. Currently, the `pipeline_map` applies to the subsequent loop, linking it to one loop variable, which means there is only one `split_iter` for each OpenMP region. Future work will extend it to support nested loops.

Figure 2 shows a three-level nested loop that performs a stencil computation in which `pipeline(static[1,3])` sets `chunk_size` to 1 and the number of GPU streams to 3. The `to pipeline_map` clause specifies that the three-dimensional input array `A0` will be pipelined. By default, we split the outermost loop. Here we denote the outer loop variable as k. We use a function of k and `<num>` to indicate the data chunks that we must copy before launch of the kth chunk's kernel. For instance, the `[k-1:3]` indicates that we must copy the $k-1$, k and $k+1$ chunks in that dimension to the device before the kth kernel executes. The `[k-1:3]` in the first set of brackets on `A0` means we split this array by its Z dimension. It defines the dependency relationship between the array and the outermost loop. For example, before kernel iteration `k=t`, we must copy chunk `t-1`, `t`, and `t+1` of `A0` to device memory. The `[0:ny-1][0:nx-1]` defines the other dimensions of array `A0`. The `from pipeline_map` clause defines the output array `Anext`. For this array, the `[k:1]` indicates that each iteration only stores its corresponding chunk. The `teams distribute parallel for` clause still parallelizes the nested loop i and j inside loop k.

A powerful code analysis engine capable of deep analysis of code and dependencies [10] could significantly simplify our proposed extension. Potentially the compiler could determine the array definition information and even the data dependencies. However, the assumption of these capabilities would limit the applicability of our extension to code that can be analyzed completely at compile time and complicate its adoption into the OpenMP specification. Thus, our prototype allows all parameters to be passed explicitly.

IV. Implementation

Based on our proposed extension, we implement a prototype runtime framework and extend four applications to exploit it: (1) a Lattice QCD application; (2) the stencil benchmark from the Parboil benchmark suite [11]; and, from the Polybenchmark set [12], (3) the 3D Convolution benchmark; and (4) the Matrix Multiplication benchmark. We split each loop into configurable-sized chunks that are handled by different streams. Each chunk has data dependencies that must be present on the device before its kernel executes. As we already define the number of streams, chunk size, and data dependencies in our extension, we can pre-allocate a device buffer that conforms to the loop's memory usage.

We map the data from the original data space to the buffer data space and copy each chunk to its corresponding location in the buffer. Currently, we use the mod operator (%) to get the offset of each chunk inside the buffer. For example, if we have a buffer that can hold four chunks, so it has positions 0, 1, 2, and 3, then, we copy chunk `i` to position `(i % 4)`. Once a data chunk is not needed for later partitions (kernels), we replace it. As long as the data is present, we schedule the corresponding subtask kernel to launch on the GPU.

While we target OpenMP syntax, more OpenACC implementations are currently available, as mentioned in Section II. Thus, we implement a prototype on top of OpenACC. We first transform the benchmarks into OpenACC as a baseline that we denote as "Naive". We implement a pipelined version ("Pipelined") of each benchmark that manually divides the iterations but does not alter array indices. Thus, it requires the full memory footprint in device memory. Finally we use our extended runtime to map the chunks to a reduced memory space ("Pipelined-buffer").

Based on the stream numbers and array declaration information, we pre-allocate a fixed-size buffer at the beginning of execution. Each array defined by the `pipline_map()` clause in the OpenMP region is associated with a data region. The array dimensions, `chunk_size` and `num_stream` determine the size of the device buffer for this data region. Once created, our runtime records the array's information for later use. We use a static pointer for the device buffer, which we allocate in GPU memory with `cudaMalloc()` on NVIDIA devices or `clCreateBUffer()` on AMD devices. We use `cudaHostalloc()` to allocate pinned host memory, which avoids the data movement time from virtual to pinned buffer memory. The asynchronous memory copy is handled by `cudaMemcpyAsync()` for contiguous data movement; we use corresponding OpenCL functions for AMD.

We also implement a 2D array interface using `cudaMallocPitch()` and `cudaMemcpy2DAsync()` to support non-contiguous data transfer. Currently, our prototype handles non-contiguous copies for 2D arrays, which means buffering a "Block" of a matrix. If `split_iter` is applied to both dimensions of a 2D array, we mark it as a 2D data region and record the corresponding information, e.g., `x_offset` and `y_offset`. Depending on the data dependencies of each subtask, we map the required data to this buffer and then pass the offsets in the buffer to the corresponding computation kernels.

To provide a fair comparison, we keep the OpenACC kernel structure the same as our pipelined version. Because we use a pre-allocated buffer instead of an array, we can only use `deviceptr()` to point to these buffer pointers inside the computation kernels. The OpenACC kernel regions are linked to different GPU streams with the `async()` clause. However, the data of each chunk only uses part of the buffer. Unfortunately, OpenACC runtime APIs do not support this partial array asynchronous copy. Thus, we use `cudaMemcpyAsync()` or `clEnqueueWriteBuffer()` Since OpenCL uses `cl_mem` as the data type instead of a pointer, which is not compatible with `deviceptr()` in PGI's OpenACC for AMD OpenCL, we implement a small OpenCL kernel to extract the pointer from the `cl_mem` data type before passing it to the OpenACC kernel. Since we only do this procedure once at the beginning of the benchmark when we allocate the buffer, it has little performance impact. The back-end runtime generates a new the device base pointer and corresponding offsets, leaving the body identical and making compiler analysis and index transformation unnecessary. To link this data copy to the corresponding GPU stream that the OpenACC `async()` clause creates, we use `acc_get_cuda_stream()` and `acc_get_opencl_queue()`. These APIs help us to retrieve the stream used by the `async()` clause to match our manual transfers with the kernels.

We do not use the OpenACC `acc_map_data()` clause to map partial host arrays to device buffers for several reasons. First, for each host array, we map different indices to one pre-allocated device buffer in a round-robin order and use an offset within the device buffer for array accesses. Second, `acc_map_data()` can only map one segment of the host array to one device buffer. Mapping multiple host array indices to different locations in the device buffer results in an error. Third, based on our experiments, using the `acc_map_data()` API with the asynchronous update directive is slower than directly using the CUDA memory-copy APIs even without asynchronous operations.

V. Evaluation and Discussion

We evaluate our approach on four applications: a 3D Convolution benchmark; a stencil benchmark; a matrix multiplication benchmark; and a production Lattice QCD application. We run our experiments on two types of GPUs: AMD Radeon 7970 and NVIDIA Tesla K40m. Our AMD experiments run on a node with an AMD Radeon HD 7970 GPU, which has 2048 stream processors and 3GB of on-board memory. Our NVIDIA experiments run on a node containing two NVIDIA Tesla K40m GPUs, each of which has 2880 stream cores and 12GB of on-board memory.

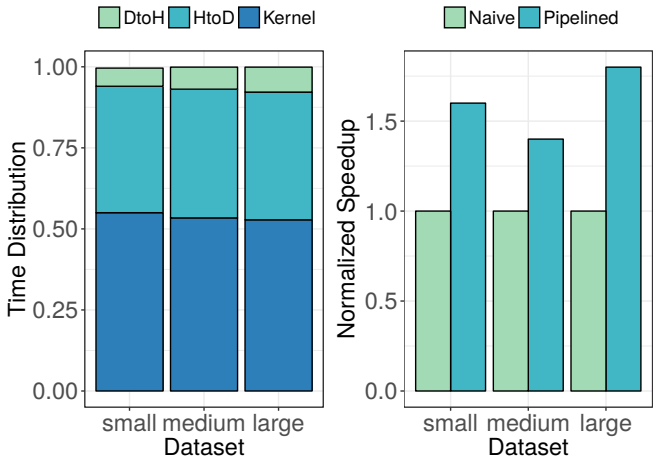

Figure 3: Lattice QCD Time Distribution (left) and Normalized Speedup (right) on NVIDIA K40m

For each benchmark, we measure the performance in terms of the function that contains the GPU operations, including all transfers but ignoring time for code that is identical in all versions. We execute all test runs six times and use their average as the final result.

A. Initial study of the pipeline technique

The naive offload model, i.e., synchronously copying and executing, is inefficient. Figure 3 shows a time distribution on an NVIDIA K40m of different phases in a naive *Lattice Quantum Chromodynamics (QCD)* application written with OpenACC. Data transfers consume nearly 50% of execution time, during which no computation is performed. This execution model wastes GPU and CPU compute resources during data transfers. Thus, the current standard interface still has limitations in terms of performance, programmability, and memory usage. To understand the pipeline technique and the impact of stream counts and data sizes, we first use the Lattice QCD application as a case study on the NVIDIA Tesla K40m GPU. From Figure 3, we observe that pipelining achieves a $1.6\times$ speedup for the small test case. As the problem size grows, the speedup increases, indicating that larger cases may approach the theoretical upper bound of $2\times$, which would be achieved if data transfers and computation were perfectly overlapped.

We also vary chunk size and number of streams in Figure 4. These two parameters can significantly affect performance. The number of streams value is the number of GPU streams that we use in parallel, which is the number of

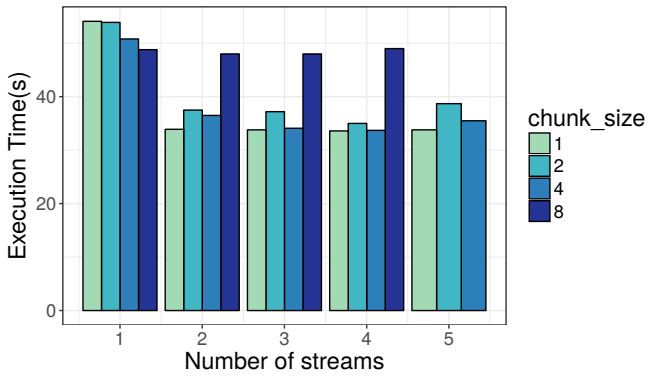

Figure 4: Different chunk sizes and GPU stream counts on NVIDIA K40m (large test case)

transfers and kernels that may simultaneously be in flight. More GPU streams could potentially hide more "bubbles" in the pipeline. However, more GPU streams requires more scheduling overhead. As we divide the task into multiple chunks, chunk size determines the size of each chunk and, thus, the number of chunks. More chunks requires more API calls, and thus more overhead. Few chunks mean that the transfers of the first input chunk and the last output chunk, which cannot be overlapped, are larger and account for more of the runtime which can degrade performance. Thus, we vary these parameters to explore the trade-off.

Figure 4 shows the results for the large test case on the K40m. Using two streams generally performs significantly better than one, showing the benefits of overlapping data transfers and computation. However, using more than four streams offers no further benefit due to increasing API and scheduling overheads while only slightly increasing potential overlap. Increasing the chunk size reduces API call and kernel launch overhead but makes load balancing harder. Increasing the chunk size usually does not adversely impact performance. Thus, we can ignore the additional overhead to use more chunks for this case. The K40m needs two streams to reach its best performance for this application.

B. 3D Convolution

Many science and engineering applications use convolutions on multi-dimensional periodic data. Applications include deconvolving blurred images, signal and image processing, noise suppression, feature extraction, wave properties modeling and many others [13], [14]. We use the 3D Convolution benchmark from the Polybenchmark set as an example on which to evaluate our approach.

Figure 5 present its performance on the K40m. The Pipelined version achieves $1.45\times$ speedup over the Naive version. Our prototype also delivers $1.46\times$ speedup over the Naive version, which provides exactly the same performance compared to the hand-coded Pipelined version.

Figure 6 shows the memory usage across versions. Since the default test case of the benchmark is relatively large, the Naive and Pipelined versions require about 3.5 GB of

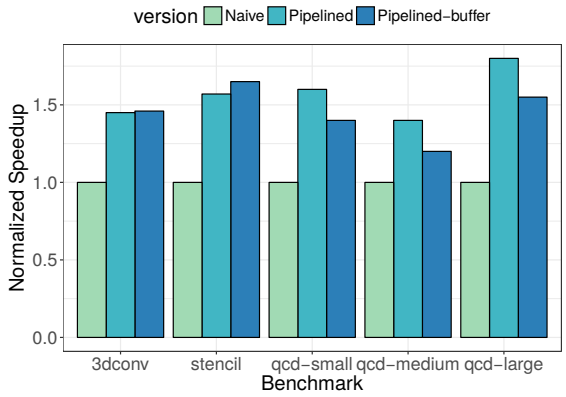

Figure 5: Performance Evaluation

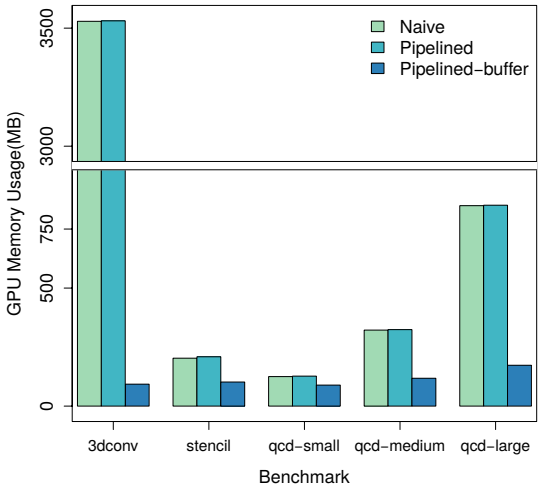

Figure 6: GPU Memory Usage Evaluation

GPU memory. Our Pipelined-buffer version only consumes 93 MB of GPU memory, which means we could save 97% of device memory. With this huge memory savings, we could potentially run much larger datasets or keep other useful data structures in device memory for a larger application.

Figure 7 shows that the number of overlapping streams affects the performance of the Pipelined version. However, using two streams no longer delivers the best performance; we instead need up to eight streams to achieve the best performance. As our results show with our other applications, the number of streams can significantly affect performance, but the ideal number of streams varies across applications.

We also find that our prototype uses slightly more memory as the number of streams increases, because we must preallocate a larger buffer as we increase the number of streams. Still, we reduce memory use 96% even with eight streams.

Figure 8 (left) shows the performance of the 3D Convolution benchmark on the AMD Radeon 7970 GPU. We first compare the Naive version with the Pipelined version. The Pipelined version is 57% slower than the Naive version,

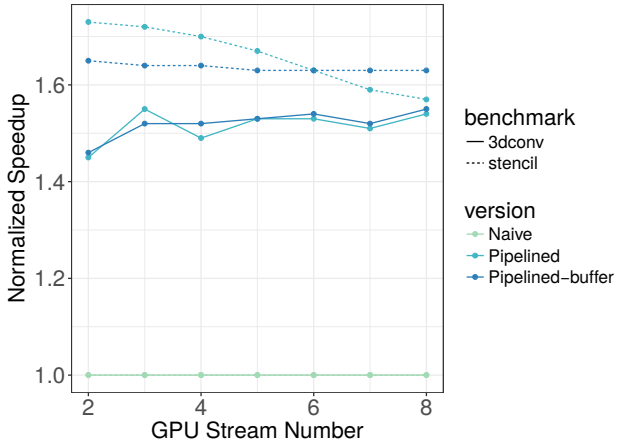

Figure 7: Execution time varying GPU stream count on NVIDIA K40m

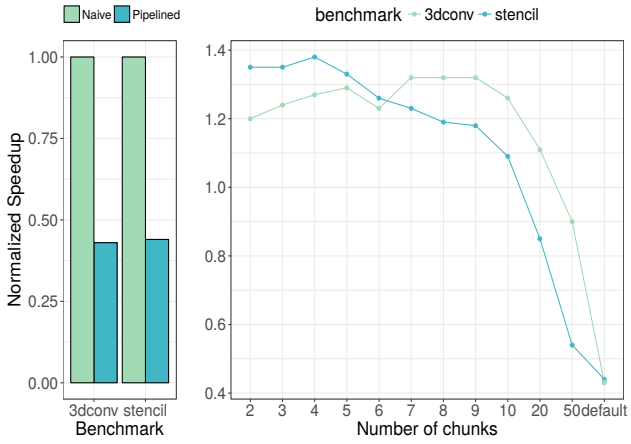

Figure 8: 3D Convolution and Stencil performance degradation (left) and normalized speedup , varying number of chunks (right) on AMD HD 7970

which is significantly different from our NVIDIA K40 results. To understand this difference, we use the AMD APP Profiler [15] to profile the Pipelined version, which reveals that data transfer times lead to the significant performance degradation. Although the data volume that is transferred is the same, the Pipelined version takes much longer to move it: the transfer rate for the Naive version is about 6 GB/s while it is only 2 GB/s for the Pipelined version.

To ameliorate this issue, we vary the chunk size and number of streams. Our conclusions include that even if more chunks imply more API call overhead, it can be ignored on NVIDIA GPUs. However, that overhead is more significant with the AMD GPU. The AMD APP Profiler results indicate that the performance degradation arises because:

- We split the task by the outer loop into small chunks, which means the chunk size is 1 and number of chunks is the problem size in that dimension, which requires many API calls and high scheduling overhead;
- Splitting the tasks into small chunks decreases the array size of each transfer, possibly to below the data transfer unit size for the AMD GPU, thus limiting bandwidth.

To test our theory, we modify our code to decrease the number of chunks. We then evaluate the performance of the Pipelined version versus the Naive version as we vary the number of chunks. Figure 8 (right) shows that if we split the problem into only two chunks that we achieve $1.2\times$ speedup over the Naive version. Performance improves as we increase the number of chunks until we use nine chunks, after which it degrades sharply. Performance is worse than the Naive version using between 20 to 50 chunks and continues to decline to the default chunk count.

C. Stencil

The Parboil Stencil benchmark represents an iterative Jacobi solver of the heat equation on a 3-D structured grid, which can also be used as a building block for more advanced multi-grid PDE solvers. We implement a prototype of the stencil benchmark using our approach.

Figure 5 shows the performance evaluation for the stencil benchmark on the K40m GPU. The Pipelined version, which uses native OpenACC pragmas to pipeline the kernel computation and data transfer, achieves $1.57\times$ speedup over the Naive version. Our Pipelined-buffer version is even faster than the Pipelined version, even including the time to handle array indexing and function calls. Our analysis finds that we only use two streams to implement the Pipelined-buffer version. However, we assign one stream to handle each subtask with the OpenACC `async()` clause, which indicates that it uses the maximum number of available GPU streams by default. Although more GPU streams could potentially hide more bubbles in the pipeline, they require more scheduling and API calls and can create contention overhead. Overall, these effects have more overhead than the benefit from overlapping data transfer and kernel computation. Since these parameters are building blocks of our schedules, we evaluate their impact, as Figure 7 shows.

We observe that the Pipelined version uses eight (8) streams by default, which explains its execution time of 6.48 seconds in Figure 5. Further, as we increase the number of streams, the execution time of the Pipelined version increases dramatically while our Pipelined-buffer version remains stable. If we limit the number of streams to two instead of using the default eight streams, the Pipelined version performs best. However, as the stream count increases, the performance crosses over: with over six streams, the Pipelined-buffer version is faster. Either pipelined version provides at least $1.5\times$ speedup over the Naive version.

Figure 6 shows the memory usage of our prototype for the Stencil benchmark. Our Pipelined-buffer version reduces memory consumption nearly 50% compared with the Pipelined version. Further, the GPU runtime and scheduler, rather than the data set, consume a large portion of the memory for this small test case.

The stream count can significantly affect memory use. The Pipelined version requires more memory to schedule the streams and to maintain the corresponding information. Our Pipelined-buffer version also requires a larger buffer allocation. Also, memory use increases slightly as we increase the stream count. Our approach always reduces memory consumption nearly 50% for the Stencil benchmark.

Figure 8 (left) shows the poor performance of Parboil Stencil on the AMD HD 7970 with the default number of chunks. For the Stencil benchmark, the Naive version is 56% faster than the Pipelined version. We again verify that reduced effective transfer bandwidth leads to the performance loss. Figure 8 (right) shows that with two chunks, the Pipelined version achieves $1.35\times$ speedup over the Naive version. As we increase the number of chunks to four, performance improves slightly. With more chunks, performance degrades until it is the same as the Naive version between 10 and 20 chunks, after which it becomes worse. The results with the 3D convolution and Stencil benchmarks demonstrate that data transfer bandwidth and API overhead limit the benefit of pipelining on the AMD GPU. More chunks require more API calls and scheduling overhead and reduce the chunk size below that required to maximize data transfer bandwidth.

Overall for the Stencil benchmark, our approach significantly reduces memory use while performing competitively with a hand-coded OpenACC solution. Further, our approach automates index translation and scheduling, which improves programmability, thus increasing the key motivation to use directive-based extensions. We find that stream count can impede the OpenACC solution. Using too many GPU streams reduces performance of the Pipelined version, while our prototype is not sensitive to stream count.

D. Lattice QCD

Quantum Chromodynamics (QCD) is the component of the standard model of elementary particle physics that governs the strong interactions. Our Lattice QCD benchmark is a larger application from the SciDAC Lattice Group. The main computational subroutine has several parallel regions, which operate on a high-dimensional lattice. These features complicate a hand-coded implementation, which indicates that programmability is particularly important. The problem size can be formalized by $O(Cn^4)$ where C is a relatively large constant. We evaluate our prototype with three data sets: n=12 (small), 24 (medium), and 36 (large).

Figure 5 shows the performance of the Lattice QCD code. In the large test case, our prototype delivers $1.54\times$ speedup over the Naive version. The huge indexing operation to map the high-dimensional space to the pre-allocated buffer probably leads to the performance difference. We pass the offset variables into the OpenACC kernel region to point to the corresponding location inside the pre-allocated buffer. Since the kernel is much larger and contains many more array element accesses, the index calculations, additional operations inside the kernel, reduce performance compared to the hand-coded version. Nonetheless, the Pipelined-buffered version significantly outperforms the Naive one.

Figure 6 shows that compared with the Pipelined version, our prototype significantly reduces memory use. As we increase the problem size, the memory savings also increase. For the largest test case, our approach reduces GPU memory use up to 79% and achieves competitive performance.

E. Matrix Multiplication benchmark

Matrix multiplication is a fundamental building block for many scientific computing applications. Moreover, the algorithmic patterns of matrix multiplication are representative. In our previous benchmarks, all data transfers are contiguous. In this section, we use the Matrix-Multiplication benchmark from Polybenchmark suite as a case study to investigate the performance of our approach with non-contiguous data transfers.

We use a naive OpenACC matrix multiplication implementation from Polybenchmark suite as our "baseline". With the matrix multiplication $A \times B = C$, this Naive implementation assigns one GPU thread to each element in matrix C. Each GPU thread gathers a line of A and a column of B and then calculates the corresponding element in C.

Many optimization methods have been developed to improve matrix multiplication performance; matrix blocking or tiling is an important one. By splitting the matrix into tiles, the size of each sub-matrix can be controlled to fit in shared memory. We assign one GPU thread block to each sub-matrix multiplication after loading the elements into shared memory. We accumulate these results into C. Ensuring shared memory use with OpenACC is difficult; we use the `private()` and `cache()` clauses. We denote this version as the "block-shared" version. Each task only needs data from a column of blocks in matrix A and a row of blocks in matrix B. We then apply our previous approach to this benchmark, partitioning the inputs and tasks into chunks by columns in A and rows in B. We assign one GPU stream to each task and copy the necessary data to a pre-allocated buffer. Mapping columns of blocks in Matrix A requires non-contiguous data transfers. After that we launch the computation kernel, and finally pipeline these GPU streams. This version is our "pipeline-buffer" version.

Figure 9 shows matrix multiplication performance across versions on an NVIDIA K40 GPU. We observe that the block-shared version, which uses block partition and shared memory, can achieve up to $3\times$ speed up over the baseline: Using shared memory significantly reduces global memory access. We also observe that our pipeline-buffer version achieves almost the same performance as the block-shared version. We then use NVIDIA Visual Profiler to profile these two versions. We find that since the matrix multiplication is compute bound, the data transfer takes little time compared

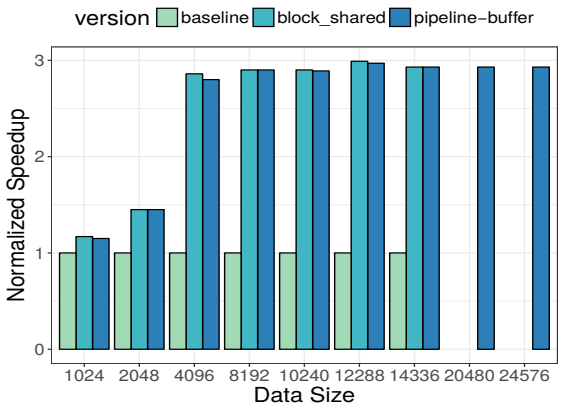

Figure 9: Matrix Multiplication performance on NVIDIA K40m

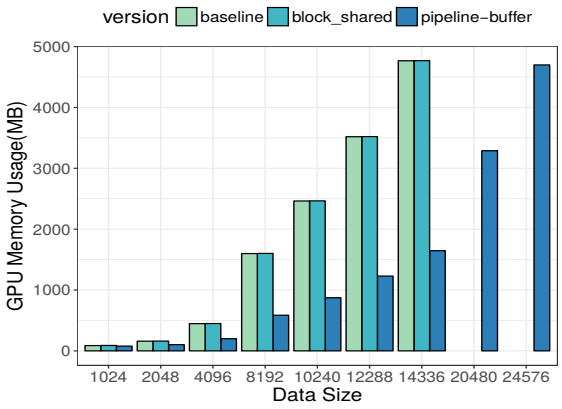

Figure 10: Matrix Multiplication memory consumption on NVIDIA K40m

to kernel computation. Although non-contiguous data transfers take more time, it can be completely overlapped with the kernel computation. Thus, the two versions achieve nearly the same performance.

Figure 10 shows the memory usage of our Matrix Multiplication versions. Pipelined-buffer significantly reduces memory consumption. As we increase the problem size, the memory savings also increase. Since we only split the Matrix A and B, if the data size is large enough, it reduces memory use nearly 66% while delivering competitive performance. This savings allow the Pipelined-buffer to compute, with no performance loss, problem sizes that exceed GPU memory for the other two versions, as shown by the two rightmost problem sizes in Figures 9 and 10.

F. Evaluation Summary and Discussion

We implement a prototype using our approach for the Parboil Stencil benchmark, the Polybench 3D Convolution and Matrix Multiplication benchmarks, and a Lattice QCD application. Our approach significantly reduces GPU memory use while delivering competitive performance to a hand-written pipelined version. The complex relationship between concurrency in data transfers, kernel launching, and stream scheduling overhead makes optimal performance difficult to achieve with hand-coded approaches. The trade-off does not have a constant solution but choosing the wrong values can adversely impact performance.

In terms of memory consumption, our implementation can save a huge portion of GPU memory for these benchmarks. Our results demonstrate that we can save more memory as the size of the test case increases. This increase in savings is because splitting the task into multiple sub-tasks decreases the space complexity of the code by one dimension. For example, we decrease the space complexity of the Lattice QCD application from $O(n^4)$ to $O(Cn^3)$, where C is a coefficient related to the dimension that we split. Thus, we can save more memory space with a larger split dimension,

The implementation of our prototype revealed limitations of OpenACC for this pattern. The naive offloading model, synchronously copying and executing in sequence, is inefficient. However, manually pipelining the kernel computation and data transfer significantly reduces programmability. Moreover, this approach allocates GPU memory based on the host array size; no ”partial array asynchronous copy” APIs are available. Thus, regardless of the use of synchronous or asynchronous copies, we must allocate the entire array in the GPU memory, exactly the same size as on the host. Our approach, by mixing CUDA and OpenACC APIs, handling the indexing to map the host array to a pre-allocated device buffer, and scheduling data movement and kernels correctly, addresses this problem. Our proposed extension improves programmability while achieving high performance and significant reductions in memory use.

We also find that the AMD GPU is sensitive to the number of chunks that we create, unlike NVIDIA GPUs. Using more chunks suffers from more API calls and scheduling overhead. Moreover, if the chunk size is too small, the data transfer does not achieve full bandwidth.

We also investigate the feasibility of integrating non-contiguous data transfers into our approach. Our results shows that although the non-contiguous data transfers take much longer, they can be perfectly overlapped with compute bound kernels like matrix multiplication. Our approach could significantly reduce memory usage while delivering competitive performance in this case.

Our prototype results show that directive-based programming models for accelerators should include our partitioning and pipelining extension. This approach significantly improves performance and programmability. It also supports running applications with huge data sizes without complex coding changes.

VI. Related Work

In this paper, we design a method to support compiler-implemented pipelining for data transfer and compute overlap in directive-based models such as OpenMP [2] and OpenACC [4]. Our preliminary work [16] presents some benefits of this approach. While double buffering, or pipelining in general, is a common manual optimization, it is

not a common facility of either production programming models or research prototypes although some have explored mechanisms that could support it.

Task-based models like OmpSs [17], [18], [19] and StarPU [20] construct graphs of "tasks" composed of statically sized chunks of data and computation, which are then scheduled. A user can achieve overlap by subdividing a given loop into a number of tasks, as long as they select the size and translate addresses manually. Our extension dynamically generates a range of logical tasks from the representation provided by the user, providing a similar result but giving the runtime more flexibility.

Higher-level logically global models like Legion [21] encode the structure of their data and computation as part of the base model and can apply optimizations like those that we discuss in their runtimes. The challenge with these is that they cannot be incrementally applied to existing codes, requiring significant refactoring if not rewriting. Similarly, logically global models like Chapel [22] support optimization of abstract loop computations through custom domain maps and other policies, but existing codes must be modified significantly to use them.

CoreTSAR [23], [24] explores automated coscheduling between devices with potentially disjoint memory spaces. CoreTSAR uses mapping functionality that associates data to computation along a single dimension for certain specific patterns. Our specifications take similar information to the array association pattern employed by CoreTSAR. However, CoreTSAR uses this information to divide computation across devices rather than to overlap computation and communication and to reduce memory use.

Our extension maps high-dimensional arrays to a low-dimensional buffer, from non-contiguous to contiguous. Recent studies on MPI libraries such as MVAPICH2 [25], [26], [27], MPICH2 [28], [29], [30], and OpenMPI [31] provide such support to pipeline data transfers between PCIe with the data transfer on high performance interconnects to optimize bandwidth. Some of the custom data-type facilities of these libraries provide similar specification facilities to those that we propose, but differ in that they represent the data type as a whole rather than as something tied to computation and thus indexable as part of one.

At the system level, studies such as ADSM [32], CGCM [33], Spark-GPU [34], and RSVM [35] provide compiler-based optimizations for data management and movement between CPUs and GPUs, depending on static or dynamic compile-time analysis or on programmer supplied annotations. Our extension provides an interface to map data from high dimensional host arrays to low dimensional device buffers, which significantly simplifies the use and application of those optimizations.

Current compute nodes with over 256-GB system memory support large memory applications like weather forecasting. However, current GPUs only have 5GB to 12GB of discrete GPU memory, a major obstacle in porting these applications to accelerators. Beyond the size restrictions imposed by accelerators, neither OpenMP nor OpenACC have any mechanism to deal with an out-of-memory situation on the device. When one occurs, the user cannot recover from the associated error. Our extension not only improves the usability of streaming kernels on these platforms, but also codee portability by making it more resilient to changes in device memory sizes.

VII. Conclusion

In this paper we propose a directive-based pipelining extension for offload models such as OpenMP 4.X and OpenACC. Our extension allows GPU programmers to pipeline data transfers without major refactoring, thus automating overlap of computation and communication. Further, mapping subsections of the host array to a device buffer reduces memory requirements. To show the benefits of our design, we choose four applications: the Stencil benchmark from the Parboil suite, the 3D Convolution and Matrix Multiplication benchmarks from the Polybenchmark set, and a SciDAC Lattice QCD application. We extend them with our prototype runtime and present a detailed evaluation that compares the programmability, performance and GPU memory consumption of our approach to that of a naive OpenACC version.

Our results show that our extension can significantly reduce memory consumption and deliver excellent performance. The memory savings of our approach increase with problem size. Moreover, our implementation is less sensitive to the number of streams used than a typical hand-coded pipelining solution; choosing the wrong stream count can significantly degrade performance with the Pipeline version.

Our prototype implementation already shows the benefit of our directive-based pipelining extension, in terms of programmability, performance and memory consumption for some specific applications. We will continue this work by investigating more benchmarks and use-cases for our extension. Currently, our experiments use NVIDIA and AMD GPUs; we will test and analyze our approach on other systems, such as Intel Xeon Phi co-processors, and even multi-nodes with different accelerators. We are also considering a source-to-source translator based on our previous work [23], [24]. The current extension is still in the early stages in how to define dependencies. We will design a function-based extension that allows the developer to pass in a function pointer to improve that functionality. Integrating powerful compiler-level code analysis and optimization could significantly improve performance, especially for the indexing and API call overhead. Finally, we will further study how the other parameters affect our design and integrate a performance model in an autotuning scheduler.

VIII. Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] J. J. Dongarra, H. W. Meuer, and E. Strohmaier, "Top500 Supercomputer Sites," 1994.

[2] OpenMP ARB, "OpenMP Application Program Interface Version 4.5," 2015.

[3] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman, "Early Experiences with the OpenMP Accelerator Model," in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 84–98.

[4] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC–First Experiences with Real-World Applications," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 859–870.

[5] T. R. Scogland, J. Keasler, J. Gyllenhaal, R. Hornung, B. R. de Supinski, and H. Finkel, "Supporting Indirect Data Mapping in OpenMP," in *OpenMP: Heterogenous Execution and Data Movements*. Springer, 2015, pp. 260–272.

[6] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao, "cuart: Fine-grained algebraic reconstruction technique for computed tomography images on gpus," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, 2016, pp. 165–168.

[7] 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 *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 136–143.

[8] K. Krommydas, T. R. Scogland, and W.-c. Feng, "On the Programmability and Performance of Heterogeneous Platforms," in *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*. IEEE, 2013, pp. 224–231.

[9] Y. Wang, Q. Qin, S. C. W. See, and J. Lin, "Performance Portability Evaluation for OpenACC on Intel Knights Corner and NVIDIA Kepler."

[10] Y. Lin and D. Padua, "Compiler Analysis of Irregular Memory Accesses," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 157–168, 2000.

[11] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[12] L.-N. Pouchet *et al.*, "Polybenchmarks Benchmark Suite," 2013.

[13] R. C. Gonzalez and R. E. Woods, "Digital Image Processing," 2002.

[14] S. F. Boll, "Suppression of Acoustic Noise in Speech Using Spectral Subtraction," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 27, no. 2, pp. 113–120, 1979.

[15] A. Corparation, "Amd stream profiler."

[16] X. Cui, T. R. Scogland, B. R. de Supinski, and W.-C. Feng, "Directive-based pipelining extension for openmp," in *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*. IEEE, 2016, pp. 481–484.

[17] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011. [Online]. Available: http://www.worldscinet.com/abstract?id=pii:S0129626411000151

[18] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive Programming of GPU Clusters with OmpSs," *International Parallel and Distributed Processing Symposium*, pp. 557–568, 2012.

[19] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces," in *ACM International Conference on Supercomputing*. ACM, Jun. 2013.

[20] C. Augonnet, S. Thibault, and R. Namyst, "StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines," Laboratoire Bordelais de Recherche en Informatique - LaBRI, RUNTIME - INRIA Bordeaux - Sud-Ouest, Tech. Rep. RR-7240, Mar. 2010. [Online]. Available: http://hal.inria.fr/inria-00467677

[21] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE Computer Society, 2012, pp. 1–11.

[22] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[23] T. R. Scogland, W.-c. Feng, B. Rountree, and B. R. de Supinski, "CoreTSAR: Core Task-Size Adapting Runtime," 2014.

[24] T. R. Scogland, B. Rountree, W.-c. Feng, and B. R. de Supinski, "Heterogeneous Task Scheduling for Accelerated OpenMP," in *IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2012, pp. 144–155.

[25] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters," *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 257–266, 2011.

[26] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda, "Optimized Non-Contiguous MPI Datatype Communication for GPU Clusters: Design, Implementation and Evaluation with MVAPICH2," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE, 2011, pp. 308–316.

[27] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 10, pp. 2595–2605, 2014.

[28] A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-c. Feng, K. R. Bisset, and R. Thakur, "MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-Based Systems," in *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*. IEEE, 2012, pp. 647–654.

[29] J. Jenkins, J. Dinan, P. Balaji, T. Peterka, N. F. Samatova, and R. Thakur, "Processing MPI Derived Datatypes on Noncontiguous GPU-Resident Data," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 10, pp. 2627–2637, 2014.

[30] A. Aji, L. Panwar, F. Ji, K. Murthy, M. Chabbi, P. Balaji, K. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummey *et al.*, "MPI-ACC: Accelerator-Aware MPI for Scientific Applications," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 27, no. 5, pp. 1401–1414, 2016.

[31] W. Wu, G. Bosilca, S. Jeaugey, and J. Dongarra, "GPU-Aware Non-Contiguous Data Movement In Open MPI."

[32] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 347–358.

[33] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU Communication Management and Optimization," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 142–151.

[34] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang, "Spark-gpu: An accelerated in-memory data processing engine on clusters," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016.

[35] F. Ji, H. Lin, and X. Ma, "RSVM: A Region-Based Software Virtual Memory for GPU," in *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*. IEEE, 2013, pp. 269–278.