Memory Access Patterns: The Missing Piece of the Multi-GPU Puzzle

Tal Ben-Nun
Department of Computer Science, Hebrew University of Jerusalem, Israel
talbn@cs.huji.ac.il

Ely Levy
Department of Computer Science, Hebrew University of Jerusalem, Israel
elylevy@cs.huji.ac.il

Amnon Barak
Department of Computer Science, Hebrew University of Jerusalem, Israel
amnon@cs.huji.ac.il

Eri Rubin
Department of Computer Science, Hebrew University of Jerusalem, Israel
eri1@cs.huji.ac.il

ABSTRACT

With the increased popularity of multi-GPU nodes in modern HPC clusters, it is imperative to develop matching programming paradigms for their efficient utilization. In order to take advantage of the local GPUs and the low-latency high-throughput interconnects that link them, programmers need to meticulously adapt parallel applications with respect to load balancing, boundary conditions and device synchronization. This paper presents MAPS-Multi, an automatic multi-GPU partitioning framework that distributes the workload based on the underlying memory access patterns. The framework consists of host- and device-level APIs that allow programs to efficiently run on a variety of GPU and multi-GPU architectures. The framework implements several layers of code optimization, device abstraction, and automatic inference of inter-GPU memory exchanges. The paper demonstrates that the performance of MAPS-Multi achieves near-linear scaling on fundamental computational operations, as well as real-world applications in deep learning and multivariate analysis.

CCS Concepts

•Computing methodologies → Parallel programming languages;

Keywords

Multi-GPU Programming, Memory Access Patterns

1. INTRODUCTION

Multi-GPU nodes are increasingly becoming the platform of choice for scientific high-performance computing (HPC) on both workstations and heterogeneous clusters. These nodes usually consist of a host (CPU) and several GPU devices, linked via a low-latency high-throughput bus, e.g., PCI-Express or NVLink. Such interconnects allow parallel applications, particularly those with data interdependency, to efficiently exchange data, taking advantage of the increased computational power and memory size.

The main challenges of multi-GPU application development are load balancing and overlapping memory transfers with computations. Naïve programming, including porting single GPU code, usually results in suboptimal device utilization. Therefore, it is common practice to implement application-specific management systems in order to maximize the concurrency. Developing such multi-GPU systems is subject to complex index computations, improper load balancing and inefficient memory transfers. Examples of applications with such management systems include CFD [26, 30], medical imaging [32] and deep learning [6, 18]. Even though these applications originate from different disciplines, they ultimately share the same workload partitioning principles and memory access patterns.

An alternative approach for multi-GPU programming is to provide developers with general-purpose libraries, building upon concepts such as algorithmic skeletons [13], compiler analysis [17] and segmented containers [27]. Due to their generic design, these libraries exhibit loss of programming flexibility, overall runtime degradation and limited scalability across multiple GPUs.

This paper presents the Memory Access Pattern Specification Multi-GPU (MAPS-Multi) partitioning and device-level optimization framework. The framework addresses the above shortcomings using a novel approach, based on classification of parallel applications to input and output memory access patterns. MAPS-Multi consists of host- and device-level components, providing programmers with an easy-to-use API that allows kernels to run on multiple GPUs without modification, using programming hints. The host-level component is responsible for multi-GPU kernel partitioning, automatic memory allocation, scheduling and exchanging boundaries. The device-level component provides an iterator-based interface, which allows the programmer to access data in an index-free fashion. This component automatically performs advanced optimizations such as global

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807611

memory write caching and instruction-level parallelism.

We show that the framework relies on explicit specification of memory access patterns, which were proven to be efficient on single GPUs [24]. The framework can easily be extended to new memory access patterns, does not require other runtime environment components, and is self-contained within a header-only library. Furthermore, the framework can partition unmodified kernels, such as CUBLAS [9] routines, by implementing wrappers that specify the underlying memory access patterns. In such cases, information exchange between GPUs is performed automatically.

The main contributions of this paper are:

- We introduce a programming paradigm for multi-GPU kernel partitioning using input and output memory access patterns.
- We present a framework that eases the development of multi-GPU applications by performing kernel partitioning and boundary exchanges automatically.
- We show that by introducing output memory access patterns to kernels, advanced code optimizations can be performed.
- We present the performance and scaling of sample applications, including stencil computation, histogram calculation, and matrix multiplication, scaling up to $\sim$3.94$\times$ with 4 GPUs.
- We demonstrate the scaling of real-world problems in deep learning with convolutional neural networks and non-negative matrix factorization, achieving $\sim$2.79$\times$ and $\sim$3.17$\times$ speedups with 4 GPUs respectively.

2. MULTI-GPU PROGRAMMING

In nodes with multiple GPU devices, each device contains a fixed set of multiprocessors and a global RAM unit. GPU code (*kernels*) run on the multiprocessors in parallel by scheduling a *grid* of many threads, grouped to *thread-blocks*. Within each thread-block, which is assigned to a single multiprocessor, threads can synchronize (using local barriers) and communicate via fast shared memory.

Inter-GPU and host-device data exchanges are performed over a low-latency high-throughput interconnect, e.g., PCI-Express or NVLink. To support such exchanges, modern GPUs are equipped with multiple memory copy engines that allow simultaneous two-way memory transfer. GPU development kits provide access to these engines, as well as the compute engine (responsible for running kernels), by allowing the programmer to create several command queues per device, called *streams*.

Peer-to-peer memory transfers between GPUs can either be instigated by the host or by a device. Host-initiated memory access is performed by explicit copy commands; whereas device-initiated memory access is supported by virtual addressing, which maps all host and device memory to a single, NUMA-based address space. Although the latter method is easy to program, it may incur high overhead due to the multitude of memory transfers in small chunks over the interconnect. Therefore, optimized multi-GPU applications typically invoke large, host-initiated peer-to-peer memory transfers.

2.1 Programming Paradigm

Efficient memory access is a major programming hurdle in the development of high-performance GPU and multi-GPU applications. In such applications, programmers are compelled to manually manage indices and memory transfers, resulting in a lengthy and error-prone code.

We now present a novel, memory-oriented parallel programming paradigm for efficient utilization and automatic partitioning of GPU applications to multiple devices. The paradigm consists of many-threaded *Task*s; where each thread accesses disjoint N-dimensional data structures, called *Datum* objects, using programmer-provided input and output memory access patterns per datum. Below, we describe the abstractions on which an automatic partitioning framework can be built.

In our paradigm, task partitioning to multiple GPUs is performed by evenly distributing the thread-blocks among the devices. Using the provided memory access patterns, the datum objects are partitioned to (possibly overlapping) subsegments according to the requirements of the thread-blocks on each device.

Dependencies between tasks are defined in the traditional manner — when the output data of one task are the input data of another. To ensure that all the required memory is available to a task, the datum segments may either be transferred from the host or exchanged between the devices. By monitoring the location of the datum segments (according to the provided access patterns), these transfers can automatically be inferred.

Collection of task results to the host is performed by gathering the segments from the GPUs, subject to post-processing as required by the output access patterns. Since host memory management is not a part of the paradigm, each datum is bound to an existing host buffer.

Within a task, data access and storage are abstracted from the programmer. Internally, these abstractions can be implemented using different schemes (e.g., shared memory, registers, vectorization), specifically tuned to each architecture, in order to provide each thread with its required data. As a result, the programmer interfaces with the memory via index-free, thread-level objects that can either be accessed sequentially (using iterators) or by relative coordinates, as in N-dimensional stencil operators.

In the rest of this paper, we propose a classification of GPU memory access patterns, and then present a framework that implements the above paradigm for multi-GPU nodes.

3. PATTERN-BASED PARTITIONING

This section provides an overview of previously researched input memory access patterns, and presents a novel complementary classification of output memory access patterns.

3.1 Input Memory Access Patterns

A classification system for parallel algorithms based on input memory access patterns, with emphasis on the GPU memory hierarchy, was presented in [24]. By mapping these classes to Berkeley's "Parallel Dwarfs" [2], it was shown that this classification can be used to represent most of the existing parallel algorithms. It was also shown that the use of input memory access patterns and index-free iterators produces kernels that perform comparably to manually optimized GPU applications.

Table 1: Input Memory Access Patterns

Access Pattern	Thread Requirements	Typical Examples
Block (1D)	Each thread requires the entire buffer, loaded to thread-blocks in chunks.	All-pairs N-body simulation
Block (2D)	Each thread-block requires multiple rows of the buffer, loaded in horizontal tiles.	Matrix multiplication (first matrix)
Block (2D - Transposed)	Each thread-block requires multiple columns of the buffer, loaded in vertical tiles.	Matrix multiplication (second matrix), Matrix transposition
Window (ND)	Each thread-block requires a spatially local ND window, with information overlap between threads.	N-dimensional convolution, Stencil operators
Adjacency	Sporadic access of a dense data structure with a fixed pattern. Used in sparse matrix and graph operations.	Sparse matrix-vector multiplication, Cloth simulation
Traversal (DFS, BFS)	Each thread operates on neighbors of a vertex.	Barnes-Hut N-body algorithm
Permutation	Each thread-block loads a contiguous block of data and distributes it to the threads in a permutation.	Fast Fourier transform
Irregular	Patterns that cannot be determined in advance.	Finite state machines

The input access patterns that were identified to be prominently used in GPU applications are listed in Table 1 (column 1). Corresponding thread requirements and typical examples are listed in columns 2 and 3 respectively. These patterns were implemented in the Memory Access Pattern Simplification (MAPS) framework [21]. MAPS is a device-level abstraction that facilitates memory storage and access on GPUs, while hiding the underlying architecture-dependent read/write optimizations.

3.2 Output Memory Access Patterns

To complement the above classification system, we categorize parallel algorithms according to their output memory access patterns. This categorization is based on all possible mappings between the number of threads and the number of outputs for a single datum, as well as the structure of the output. The output patterns can thus be classified into five distinct groups:

- **Structured Injective**: In this pattern, each thread is mapped to a fixed number of distinct output indices. These indices coincide with the work dimensions of the kernel in a predictable manner, e.g., as in matrix multiplication.
- **Unstructured Injective**: This pattern corresponds to the previous mapping. However, the output datum indices in this group are uncorrelated to the thread indices, lacking spatial locality (as in FFT). As a result, this group requires duplicate copies of the entire datum in each GPU, as well as an additional post-kernel step that aggregates the scattered data.
- **Reductive (Static)**: Implements a many-to-one mapping, where the number of outputs is predetermined, regardless of the number of threads (e.g., histogram computation). In this group, data duplication and aggregation are required in order to gather the results.
- **Reductive (Dynamic)**: Corresponds to kernels with fewer outputs (determined at runtime) than threads, requiring data duplication and aggregation. One example is predicate-based filtering of arrays, where the aggregation process appends the results from each GPU to a single output array.
- **Irregular**: This group corresponds to kernels in which there is an unknown number of outputs per thread (e.g., ray-tracing).

Observe that in the special case of *Structured Injective*, it is possible to determine and allocate the exact datum segments required by each device, conserving memory usage.

4. THE MAPS-MULTI FRAMEWORK

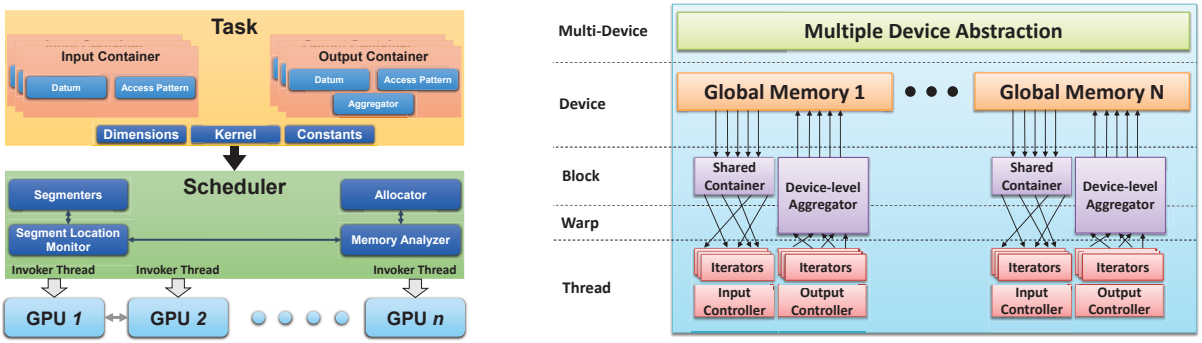
The MAPS-Multi framework is designed for automatic partitioning of kernels to multiple GPUs based on their memory access patterns. This section presents the components of the framework's host- and device-level infrastructures (shown in Figure 1), as well as an extension of the framework for running unmodified kernels on multi-GPU nodes.

MAPS-Multi, which is implemented in standard C++ over CUDA, extends the MAPS framework [21, 24] by combining its device-level optimizations with multi-GPU data transfer and boundary condition management.

The host-level infrastructure, shown in Figure 1a, implements the programming paradigm described in Section 2. The *Task* construct, shown in the upper part of the figure, is a user-provided tuple, consisting of input and output containers, each with its datum object and access pattern. Tasks also contain kernel code; grid dimensions; and constant inputs, which are fixed-sized parameters needed by all the GPUs (e.g., computational factors).

When a task is submitted to the *Scheduler* (lower part of Figure 1a), the per-GPU memory allocation requirements of each datum are inferred by the *Memory Analyzer*. Then, *Segmenter* classes, implemented for each access pattern, determine the partitioning of the task. After that, the scheduler allocates the required buffers on the GPUs using the *Allocator*; and the *Segment Location Monitor* computes the necessary inter-GPU data transfers (e.g., boundary exchanges). Finally, actual copy and execution commands are queued to each device concurrently using the *Invoker Threads*.

The device-level infrastructure, shown in Figure 1b, provides the programmer with the aforementioned index-free kernel programming interface. This interface consists of thread-level *controllers* that create input and output *iterators*, implementing the standard C++ API. Internally, the iterators access global GPU memory via *Shared Contain-*



(a) Host-level infrastructure

(b) Device-level infrastructure

Figure 1: The MAPS Multi-GPU framework

ers and *Device-Level Aggregators*, which cache the required information for input and output operations respectively. The infrastructure also maintains *Multiple Device Abstraction* that hides device management from the kernel code. This is accomplished by defining a virtual multi-GPU grid, and offsetting the thread-blocks in each device differently.

To demonstrate the components of MAPS-Multi, we implement a 2D stencil operator, which evaluates the rules of the Game of Life cellular automaton [14]. Briefly, in the Game of Life, each cell processes information from its 8 neighbors, thus implementing the *Window (2D)* input and the *Structured Injective* output memory access patterns. To conserve memory, two matrices are used as input and output buffers in alternating succession (double buffering). For further references in this section, the host and the device code samples of the Game of Life are shown in Figure 2.

We note that while the MAPS-Multi implementation of the Game of Life spans 11 lines of host code (see Figure 2a), an equivalent multi-GPU application without the framework is ∼107 lines long, most of which manage allocation, memory exchanges, stream and event creation.

4.1 Host-Level API

The host-level API of MAPS-Multi is shown in Table 2. To avoid duplicate allocation of buffers on the host, datum objects are assigned to existing memory using the `Bind` method (Figure 2a, lines 8–9). Then, the `Scheduler` class provides the programmer with access to the framework components and performs the necessary device management. In the scheduler API, all tasks are first analyzed with respect to memory usage by the `AnalyzeCall` method (lines 12–13, see Section 4.2). After all tasks have been analyzed, they are scheduled by calling the `Invoke` method (line 17). Using host-level `Aggregators`, the `Gather` method (lines 23, 25) collects the data from the devices to the host.

4.2 Automatic Memory Allocation

To support partitioning, buffers must be allocated in each device separately. This can either be accomplished by pre-allocation of the entire datum in each device; by runtime allocation of sub-segments upon task invocation; or by pre-allocation based on the memory requirements of the datum on each device. The first approach is inefficient in terms of memory consumption, whereas the second approach may create fragmented memory and requires multiple allocation calls at runtime.

Table 2: Host-Level API

class Datum	**Method Description**
`Bind`	Registers a host buffer
class Scheduler	**Method Description**
`AnalyzeCall`	Uses memory analyzer to forward-declare a task
`Invoke, InvokeUnmodified`	Schedules and calls a task
`Gather, GatherAsync`	Gathers buffers to host
`Wait, WaitAll`	Waits for specific (or all) tasks to finish executing

The MAPS-Multi memory analyzer implements the third approach. Using the explicit memory access pattern specifications, it can determine the exact amount of memory required in each device. Furthermore, it allocates the necessary memory once, creating contiguous buffers. In Figure 2a, the memory analyzer is invoked in lines 12 and 13.

To trigger the memory analyzer, the programmer is required to use the `AnalyzeCall` API method for each task, prior to the invocation of all dependent tasks. This method accepts the same parameters as the `Invoke` method. Internally, the memory analyzer tracks the current memory requirements for each datum based on its access patterns and, as in [23], computes the N-dimensional bounding box, containing both the currently stored and the predicted requirements.

Figure 3 shows the memory analysis of the Game of Life example. As explained earlier, the implementation consists of two alternating matrices. This double-buffering scheme requires two `AnalyzeCalls`: one with matrix A as input (using *Window (2D)*) and matrix B as output (using *Structured Injective*); and the other in reverse order.

On the left side of the figure, after the first `AnalyzeCall` (Figure 2a, line 12), the matrix A (according to the output pattern for B) requires allocation of four equal segments with extra space for boundaries; whereas B only requires four equally-sized segments, without boundaries. When the method is called again (right-hand side of Figure 3, corresponding to line 13 of Figure 2a), A becomes an output

```
1  typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
2  typedef StructuredInjective<T,2,ILPX,ILPY> SMat;
3
4  // Define data structures to be used
5  Matrix<T> A (width, height), B (width, height);
6
7  // Use existing host buffers as matrices
8  A.Bind(host_A);
9  B.Bind(host_B);
10
11 // Analyze memory access patterns for allocation
12 sched.AnalyzeCall(Win2D(A), SMat(B));
13 sched.AnalyzeCall(Win2D(B), SMat(A));
14
15 // Invoke the kernels
16 for (int i = 0; i < iterations; ++i)
17   sched.Invoke(GameOfLifeTick,
18                Win2D((i % 2) ? B : A),
19                SMat((i % 2) ? A : B));
20
21 // Gather processed data back to host
22 if ((iterations % 2) == 0)
23   sched.Gather(A);
24 else
25   sched.Gather(B);
```

(a) Host code

```
1  template <typename T, int ILPX, int ILPY>
2  __global__ void GameOfLifeTick MAPS_MULTIDEF(
3      Window2D<T,1,WRAP,ILPX,ILPY> current_gen,
4      StructuredInjective<T,2,ILPX,ILPY> next_gen) {
5
6    MAPS_MULTI_INIT();
7    typedef Window2D<T,1,WRAP,ILPX,ILPY> Win2D;
8    __shared__ Win2D::SharedData sdata;
9    current_gen.init(sdata);
10   next_gen.init();
11
12 #pragma unroll
13   MAPS_FOREACH(nextgen_iter, next_gen) {
14     int live_neighbors = 0, is_live = 0;
15
16 #pragma unroll
17     MAPS_FOREACH_ALIGNED(iter, current_gen,
18                          nextgen_iter) {
19       // Set variables according to the rules
20     }
21     int result = GameOfLifeConditions(...);
22     *nextgen_iter = result;
23   }
24   next_gen.commit();
25 }
```

(b) Device (kernel) code

Figure 2: MAPS-Multi code sample for the Game of Life

container that requires less memory, and thus its memory allocation remains unchanged. However, B becomes an input container; and the memory analyzer determines that the allocation for B should take the boundaries into account as well. In both cases, the 2D window size, which is 3×3 in our example (defined as a radius of one element in the second template parameter, line 1 of Figure 2a), determines the size of the boundary allocation.

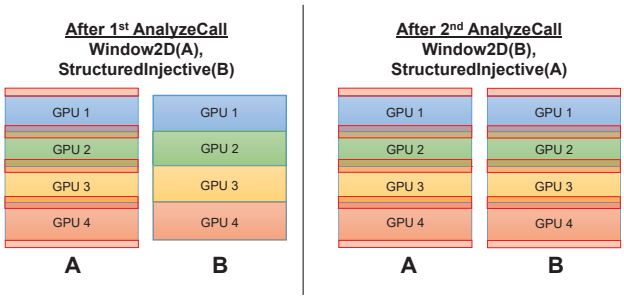


Figure 3: Memory analysis of the Game of Life

We note that while error checking is implemented in the memory analyzer, it is assumed that the programmer-provided access patterns match the task invocation parameters. Otherwise, a framework runtime error could occur when insufficient memory is allocated.

4.3 Multi-GPU Scheduler

The scheduler is the main component of the host-level infrastructure. It is responsible for mediating between the framework and the devices, invoking kernels and queuing GPU commands (e.g., memory transfers). In Figure 2a, it functions both as the API entry point (lines 12, 13, 23, 25) and as the task invoker (line 17).

The scheduler manages multiple threads, one per device. Each thread queues commands to its designated device in order to allow concurrent memory copies and kernel execution. Synchronization is managed via GPU streams and CPU thread barriers.

Algorithm 1 depicts the execution pipeline of the scheduler. When tasks are submitted to the scheduler, the algorithm gathers the required information for partitioning and invocation, using *Segmenters*, the *Segment Location Monitor* and the *Memory Analyzer*.

Algorithm 1: Scheduling Pipeline

Input: Function invocation
Output: Task handle
Construct task from function call.
Determine grid segmentation strategy according to patterns.
foreach *container* **do**
    Use Segmenter to infer memory segmentation.
end
foreach *device* **do**
    foreach *container* **do**
        Obtain allocated memory from the Memory Analyzer.
        Compute required segment copies using Segment Location Monitor.
        Offset pointers and data structures.
    end
end
Distribute memory copies to threads.
foreach *device* **do in parallel**
    foreach *segment copy* **do**
        Signal streams to wait for "segment ready" event.
        Copy segment from one device to another, aggregating as necessary.
    end
    Queue kernel or unmodified routine call.
    Record "segment ready" event for dependent tasks.
end

The algorithm begins by constructing a `Task` object from the function call (lines 1–2) and determining the partitioning scheme, explained in Section 2. Then, the memory segments for each task parameter are determined using the container `Segmenter` classes (lines 3–5). In lines 6–12, the scheduler uses the *Memory Analyzer* to obtain the GPU pointers, allocating new buffers if necessary. It then determines the necessary memory copies using the *Segment Location Monitor*. After that, the peer-to-peer memory copy commands are sent to the GPU invoker threads (line 13) to maximize data exchange concurrency. Finally, the segments are copied (lines 15–18) and the kernels are queued to the devices (lines 19–20), managing the CPU threads, GPU streams and events, to ensure memory consistency.

4.4 Segment Location Monitor

The segment location monitor tracks all host and device instances of each datum, keeping two lists of location entries per datum. The first list, `lastOutput`, keeps, for each datum segment, the location (host/device) and the datum state (e.g., in a single location, segmented or pending aggregation). The second list, `upToDate`, tracks all instances of unmodified segments in order to avoid unnecessary memory transfers.

When the scheduler determines that a segment is required by a certain device, the location monitor computes which memory segments have to be copied, and their current locations. This procedure, depicted in Algorithm 2, creates a list of segment copy operations, each of which results in a series of peer-to-peer memory copies.

Algorithm 2: Location Monitor Dependency Computation

Input: Datum, Segment, Target GPU, *lastOutput, upToDate*
Output: Segment copy operation list, *needsAggregation*

```
1  needsAggregation ← false;
2  if Datum buffer segment on target GPU is up to date then
3      return
4  end
5  if Datum up-to-date buffer is on a single location then
6      Add copy operation from current location to target GPU.
7      return
8  end
9  foreach device ≠ Target GPU do
10     Compute N-dimensional intersection between required
       segment and the lastOutput of the datum on device.
11     if intersection is not empty then
12         Add copy operation of the intersection from device to
           target GPU.
13     end
14 end
15 if lastOutput indicates that datum needs aggregation then
16     needsAggregation ← true;
17 end
```

In the algorithm, the location monitor first checks if there is an up-to-date instance of the segment on the target device (lines 2–4). If not, the segment copies are determined according to the datum state: if the whole segment resides in one device, it is copied directly (lines 5–8); otherwise, the datum is segmented among multiple devices and must be copied in parts. In this case, N-dimensional rectangular intersections are performed between the required segment and the `lastOutput` segments on each device (line 10), in order to determine the regions to copy to the target device (line 12). Upon completion of the copies, the `upToDate` list is updated with the newly copied segments in order to conserve memory copies in subsequent tasks.

Note that since the number of devices in each node is relatively small (< 10), a naïve $O(g)$ method is used to compute the intersections with all g devices.

4.5 Device-Level Optimizations

Using input containers, the MAPS framework [24] optimizes GPU global memory reads automatically. Output containers, implemented in the device-level infrastructure of MAPS-Multi, allow the incorporation of additional single GPU optimization capabilities. This section presents two such optimizations.

4.5.1 Instruction-Level Parallelism

Instruction-level parallelism (ILP) optimizations are implemented in kernels by designating each thread to process a predetermined number of elements, allowing the GPU to pipeline instructions and transfer global memory to threads in larger portions. To enable ILP optimizations, output containers provide additional template parameters that allow the programmer to specify the number of processed elements per thread. Based on the memory access patterns, the `Segmenter` ensures that the required memory is available to the kernel on each device.

The ILP device-level optimizations are shown in Figure 2b. Similarly to input containers, each output container creates device-level iterators. Using the output container ILP template parameters (line 4, parameters 3 and 4), the number of elements processed by each thread is determined. The corresponding input container is provided with the same parameters (line 3, parameters 4 and 5) in order to perform memory caching optimizations.

During runtime, the input container's thread-level iterator is aligned with the output container. This is implemented by passing the output container to the `align` method of the input container. Internally, the implementation of the input container uses the output container to offset the iterator index to the currently processed element. During compilation, the iterator methods are inlined and the output loop is unrolled, allowing the compiler to assign elements to registers and reorder instructions. To simplify this process, the `MAPS_FOREACH` (Figure 2b, line 13) and `MAPS_FOREACH_ALIGNED` macros (line 17) create output and aligned input iterators respectively, and loop over the elements automatically.

4.5.2 Device-Level Aggregators

Device-level aggregators, which are part of the output containers, perform global memory write optimizations by mediating between the threads and the global memory. These aggregators transparently reorder and unify global memory writes, as well as conserve atomic operations.

To use device-level aggregators, the programmer should call the `commit` method of the output container interface, after processing all elements (Figure 2b, line 24). Internally, aggregators are specifically tuned for each device architecture, utilizing different methods where applicable, e.g., using shared memory atomic operations in the *Reductive* patterns to store intermediate results. When `commit` is called, the output array is written to the global memory in a single, coalesced operation for each thread-block, rather than separately by each thread.

4.5.3 Device-Wide Reduction

To demonstrate that the MAPS-Multi device-level API can also be used optimize reductive operations, an implementation of histogram computation is shown in Figure 4.

```
1  template<typename T, int BINS, int ILP>
2  __global__ void HistogramKernel MAPS_MULTIDEF(
3      Window2D<T, 0, NO_CHECKS, ILP> image,
4      ReductiveStatic<int, BINS, ILP> hist) {
5
6    MAPS_MULTI_INIT();
7    __shared__ decltype(image)::SharedData s_in;
8    __shared__ decltype(hist)::SharedData s_out;
9    image.init(s_in);
10   hist.init(s_out);
11
12 #pragma unroll
13   MAPS_FOREACH(hist_iter, hist) {
14     auto image_iter = image.align(hist);
15     auto bin = *image_iter;
16     hist_iter[bin] += 1;
17   }
18   hist.commit();
19 }
```

Figure 4: Histogram kernel code

In the code, the input access pattern is a 1×1 *Window (2D)* (line 3), and the output access pattern is *Reductive Static* (line 4). Both patterns use the `ILP` parameter as the number of elements processed by each thread. The loop that controls the automatic ILP optimizations is shown in line 13. In the loop, the function call in line 14 creates an input iterator that is aligned with the current ILP index of the output container. The value of the current pixel is obtained in line 15, and in line 16 the histogram algorithm increments the output iterator. Finally, the results are committed to the global memory (line 18). We note that unlike libraries that provide pre-written kernels, the flexibility of MAPS-Multi simplifies the development of more complex algorithms (e.g., performing data transformation prior to bin incrementation).

4.6 Unmodified GPU Routines

In some cases, it is preferable to run existing, highly optimized GPU routines, rather than MAPS-Multi container-based kernels. Examples include the widely used CUBLAS [9] and CUFFT [11] single-GPU libraries, which contain manually tuned device code for each architecture.

MAPS-Multi can run external GPU routines by using wrapper functions with a predetermined prototype. This extension allows programmers to run unmodified kernels on multiple GPUs, using the framework to derive segmentation and inter-GPU data exchanges automatically. Internally, instead of invoking kernels, the scheduler calls the host-level wrapper functions for each GPU with the device ID, stream, buffer pointers and their corresponding memory segments.

A MAPS-Multi implementation (without error checking) of the SAXPY BLAS operation over CUBLAS is shown in Figure 5. The context object (line 2) is a programmer-generated data structure that contains necessary information to run the routine. In this case, it contains the CUBLAS handle objects for each GPU. Invocation-specific parameters, such as the `alpha` constant, are retrieved using the

```
1  bool SAXPYRoutine(...) {
2    CUBLASContext *c = (CUBLASContext *)context;
3    float alpha = 0.0f;
4    GetConstantParameter(parameters[2], alpha);
5    int n = container_segments[0].m_dimensions[0];
6    cublasSetStream(c->handles[deviceIdx], stream);
7    cublasSaxpy(c->handles[deviceIdx], n, &alpha,
8                (float *)parameters[0], 1,
9                (float *)parameters[1], 1);
10   return true;
11 }
```

Figure 5: CUBLAS SAXPY routine wrapper

`GetConstantParameter` function (lines 3–4). The datum segments of each container, specific to each GPU, is retrieved (line 5) from the `container_segments` argument. This array is correlated with the `parameters` argument, which contains the GPU buffer pointers. Prior to executing the kernel, its GPU stream is set to the specified argument (line 6), and the kernel is called (lines 7–9). The next two sections show additional examples that employ unmodified routines from CUBLAS and other external libraries.

5. FRAMEWORK PERFORMANCE

This section demonstrates the performance of the MAPS-Multi framework components using three fundamental computational operations. Throughout the section, multi-GPU bar graphs show the incremental speedup of each application on all device types, using 1–4 GPUs.

Our experimental setup consists of three identical nodes, each with 4 NVIDIA GeForce GPUs, as specified in Table 3. In all nodes, two PCI-Express 3 buses directly connect pairs of GPUs, where each pair is controlled by a different CPU.

Table 3: GPU Experimental Setup

Model (Architecture)	Global Memory	Multiprocessors × Cores
GTX 780 (Kepler)	3 GiB	12×192
Titan Black (Kepler)	6 GiB	15×192
GTX 980 (Maxwell)	4 GiB	16×128

5.1 Multi-GPU Scaling

The scaling of the Game of Life, histogram and matrix multiplication (SGEMM) applications using MAPS-Multi was measured. Both the Game of Life and the histogram applications were written in MAPS-Multi, using its automatic ILP and device abstraction capabilities, whereas the SGEMM application uses unmodified GPU routines from CUBLAS.

Figure 6 shows that the histogram and SGEMM applications, which do not require inter-GPU communication, scale almost linearly on all architectures, achieving up to ∼3.94× for the histogram and ∼3.93× for matrix multiplication on 4 GPUs. In the Game of Life, which requires two-line boundary exchanges per iteration, the average scaling for 4 GPUs versus a single GPU is ∼3.68×. Observe that these results are consistent on all three platforms.

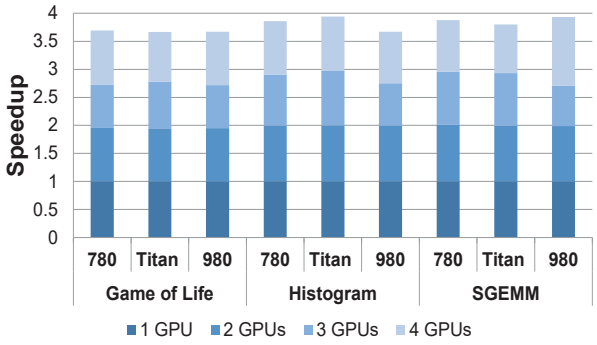


Figure 6: Framework scaling over multiple GPUs

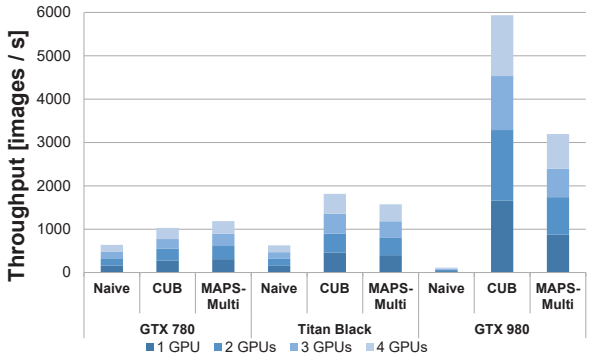


Figure 8: Histogram multi-GPU performance

5.2 Instruction-Level Parallelism

The performance of the Game of Life on an 8K square matrix using three implementation schemes is shown in Figure 7. The figure compares a naïve implementation with two implementations over MAPS-Multi, one uses shared memory optimizations and the other also includes automatic ILP optimizations with 8 elements (4 columns, 2 rows) per thread.

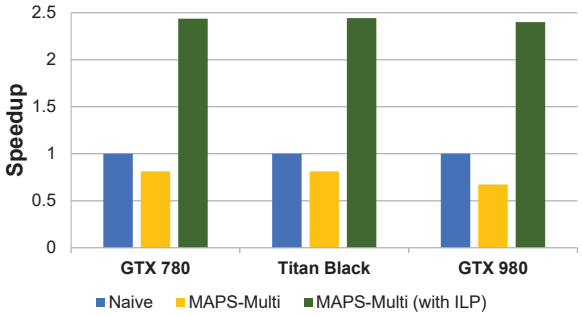


Figure 7: Game of Life single GPU performance

From the figure, it can be seen that the naïve version outperforms the non-ILP version of MAPS-Multi by ∼20–50%, depending on the architecture. This is due to the latency of accessing the shared memory for 3×3 neighborhoods, as well as the small number of required integer operations. Using ILP, however, yields a ∼2.42× performance increase over the naïve version on all architectures.

5.3 Device-Level Aggregators

We measured the throughput of device-level aggregators using a 256-bin histogram computed on an 8K square image. We compare the performance of a naïve implementation and the optimized CUB GPU library [8], both developed for a single GPU, with MAPS-Multi. In order to support multiple GPUs, the former two programs were also implemented over MAPS-Multi using unmodified routines, as described in Section 4.6.

The results of these measurements are shown in Figure 8. From the figure, it can be seen that MAPS-Multi performs better than CUB on the GTX 780. In contrast, CUB is faster on the Titan Black and more so on the GTX 980, probably due to architecture and algorithm-specific optimizations, which, by design, cannot be incorporated in the generic MAPS-Multi framework.

Observe that the naïve implementation, which utilizes global atomic operations, performs differently on the Kepler (GTX 780, Titan Black) and Maxwell (GTX 980) architectures, with corresponding runtimes of ∼6.09 ms, ∼6.41 ms and ∼30.92 ms on a single GPU. On the other hand, the runtimes of MAPS-Multi and CUB are within the same order of magnitude on all tested GPUs.

The relatively slow runtime of the naïve implementation on Maxwell can be attributed to architectural modifications, causing shared atomics to be highly preferable over global atomic operations. This is one of the main advantages of our pattern-based abstraction — the programmer does not need to be informed of such changes, nor re-tune the code for each GPU model.

5.4 Unmodified GPU Routines

To demonstrate the overhead and scaling of unmodified GPU routines, we ran a chain of 1,000 multiplications of two 8K square matrices and measured the average runtime. The performance of CUBLAS over MAPS-Multi (using unmodified routines) is compared with CUBLAS-XT [9], NVIDIA's multi-GPU interface for CUBLAS.

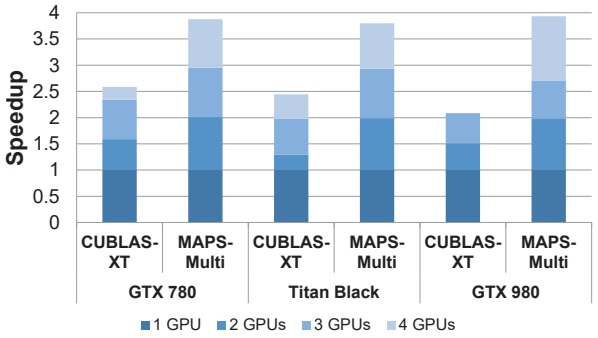


Figure 9: Matrix multiplication scaling vs. CUBLAS-XT

Figure 9 shows that the scaling of MAPS-Multi surpasses that of CUBLAS-XT on all three platforms. This can be attributed to the host-based API of CUBLAS-XT, which does not support input and output GPU buffers. Instead, each call generates host-to-device and device-to-host copy operations, slowing down GPU applications with multiple chained kernels. We note that the speedup for 4 GTX 980 GPUs is lower than that of 3 GPUs, and thus it is not shown.

Table 4 compares the single-GPU runtime of the above matrix multiplication test using CUBLAS, unmodified CUBLAS over MAPS-Multi, and CUBLAS-XT. The table shows that CUBLAS over MAPS-Multi (column 3) is only 0.2–1.3% slower than native CUBLAS (column 2). In comparison, the runtime of CUBLAS-XT (column 4) is significantly higher, due to its host-based API.

Table 4: Single-GPU Matrix Multiplication Performance

GPU	CUBLAS	CUBLAS over MAPS-Multi	CUBLAS-XT
GTX 780	365.21 ms	366.01 ms	1393.26 ms
Titan Black	338.65 ms	342.71 ms	1830.82 ms
GTX 980	245.31 ms	248.62 ms	1017.64 ms

6. REAL-WORLD APPLICATIONS

We present two real-world applications in machine learning and multivariate analysis. Each application is analyzed from the memory access pattern perspective, and its performance is compared with state-of-the-art implementations.

6.1 Deep Learning

In machine learning, multi-GPU nodes are emerging as the platform of choice [6, 18] for deep neural network training. Due to the large amount of parameters, throughput requirements and operation concurrency, such nodes are ideal for this application.

This section shows the performance of training a basic Convolutional Neural Network (CNN) called LeNet [19] on multiple GPUs with MAPS-Multi. Using the backpropagation algorithm, this CNN trains a handwritten digit classifier. To train the network, we use the MNIST dataset [20], which contains 70,000 handwritten digit images.

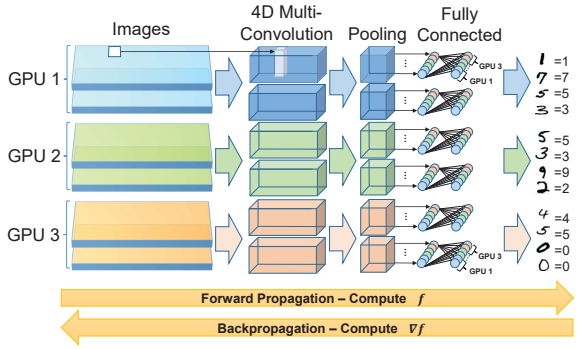


Figure 10: Convolutional neural network architecture

Figure 10 illustrates the forward and backward propagation schemes of the LeNet CNN architecture. The algorithm computes the partial derivatives of the network parameters with respect to an input image batch. The partial derivatives are, in turn, used to compute the network parameters for the next iteration, where a different image batch is loaded. The figure shows three types of operations: 4-dimensional multi-convolution, pooling and fully connected layers.

In multi-convolution, each image in the batch is separately convolved in three dimensions, using several convolution filters with different weights. This corresponds to the *Window (3D)* input memory access pattern, using the *Structured Injective* output access pattern to create a 4D tensor. The pooling operator sub-samples images by their maximal values in a local neighborhood, also implementing the *Window (3D)* and *Structured Injective* patterns. Fully connected layers are linear operators, implemented as matrix multiplication of the parameters and weights. The memory access patterns used in these layers are *Block (2D)* and *Block (2D-Transposed)*, flattening the 4D data into 2D matrices, with the *Structured Injective* output pattern.

Currently, the prevalent approach to perform deep learning on multiple GPUs utilizes data parallelism. In this approach, each GPU processes a different batch of images, exchanging partial derivatives of all the parameters during the network update phase. This approach is not scalable, as it requires each GPU to allocate space for the entire network, as well as to exchange all the parameters in each iteration.

Another approach, called hybrid data/model parallelism [18], proposes to divide the network into two parts: one containing the multi-convolution and pooling operators, and the other containing the fully connected layers. This approach partitions the first part in the same manner as data parallelism, and the second part by computing different partial derivatives of the same batch on different GPUs (see Figure 10). The second part exchanges less data, but more frequently, between the GPUs. Note that this approach partitions the parameters of the second network part among the GPUs, allowing to train large networks that do not fit in a single GPU.

Figure 11 compares the multi-GPU throughput of the hybrid approach, implemented over MAPS-Multi, using 2048 images per batch. The implementations are also compared with two state-of-the-art deep learning frameworks: Caffe [16] (rev. `2a7fe03`) and Torch [7] (rev. `288ec4b`). Since multi-GPU training is not supported in Caffe, its results are only shown for a single GPU. We note that both frameworks provide generalized solutions to deep neural networks, and thus maintain large code bases.

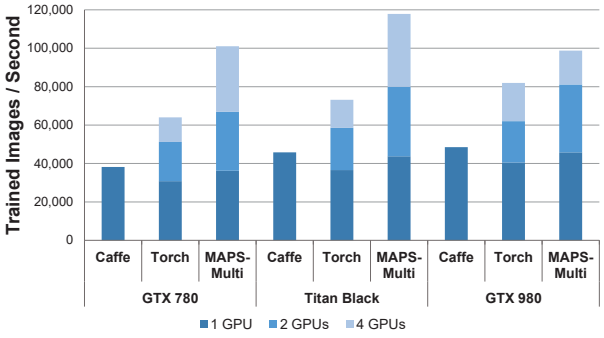


Figure 11: Deep learning performance

From the figure, it can be seen that the throughput (images per second) for a single GPU is similar in Caffe, Torch and MAPS-Multi. This is due to the fact that all the frameworks use the same routines from the cuDNN v2 [10] library. Additionally, MAPS-Multi achieves up to $\sim 2.79\times$ speedup on four GTX 780 GPUs. In comparison, on the same platform Torch achieves a speedup of $\sim 2.07\times$. Further analysis shows that this lower speedup is caused by Torch performing

all weight updates on a single GPU, as well as unnecessary device-to-host copies in each iteration.

We note that in Torch, implementing each of the above concurrency approaches requires the creation of a completely different neural network architecture. On the other hand, switching between data parallelism and the hybrid approach in MAPS-Multi requires only a single access pattern modification in the fully connected layers. The resulting speedup of the data-parallel approach using MAPS-Multi scales up to ∼3.12× on four GTX 780 GPUs, whereas the corresponding speedup of Torch is ∼2.3×.

6.2 Non-Negative Matrix Factorization

In multivariate analysis, Non-negative Matrix Factorization (NMF) performs dimensionality reduction on large, related input datasets. Formally, given a matrix $V_{n\times m}$, find two matrices $W_{n\times k}$ and $H_{k\times m}$, where $k \ll n, m$, such that $V \approx WH$.

There are many algorithms to compute the two matrices. The algorithm in this section is based on the following update rule [3]:

$$H_{ij} \leftarrow H_{ij} \frac{\sum_{p=1}^{n} W_{pi} V_{pj} / (WH)_{pj}}{\sum_{r=1}^{n} W_{ri}};$$

$$W_{ij} \leftarrow W_{ij} \frac{\sum_{p=1}^{m} H_{jp} V_{ip} / (WH)_{ip}}{\sum_{r=1}^{m} H_{jr}}.$$

Figure 12 shows the dependency graph of a single iteration of the algorithm, with respect to the memory access patterns, where each task is color-coded by its input patterns.

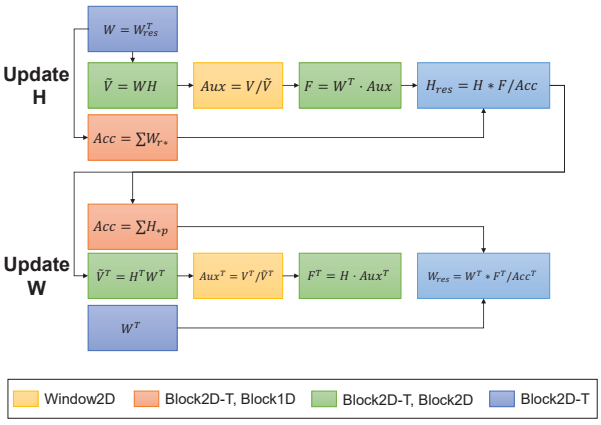


Figure 12: Dependency graph of an NMF iteration

The algorithm depicted in the figure computes H and W in alternating succession until the convergence of $\|V - WH\|$. The two above equations are represented as a chain of dense linear algebra operations, of which the computation of Acc (orange blocks in the figure) could be performed independently of the other operations, allowing kernel concurrency.

Observe that the breakdown of the update rule (in the equations) to memory-oriented tasks (in the figure) allows the matrices $\tilde{V}, Aux, F, H_{res}$ and Acc to be computed in independent stripes, without requiring a complete copy of the large V matrix in each GPU. Additionally, the inter-GPU memory exchanges, automatically inferred by MAPS-Multi, are performed twice per iteration, between the updates of H and W.

The performance of factorizing a 16K×4K matrix with $k = 128$ using MAPS-Multi is compared with NMF-mGPU [22] in Figure 13. The NMF-mGPU application is specifically tailored for multiple GPUs, containing manually optimized kernels tuned for fast NMF computation. Observe that MAPS-Multi yields higher throughput and better scalability than the NMF-mGPU application on all device types, with four GTX 980 GPUs achieving a speedup of ∼3.17×. Note that the code of NMF-mGPU consists of multiple files, spanning ∼15,000 lines, whereas the MAPS-Multi implementation consists of a single file with 870 lines.

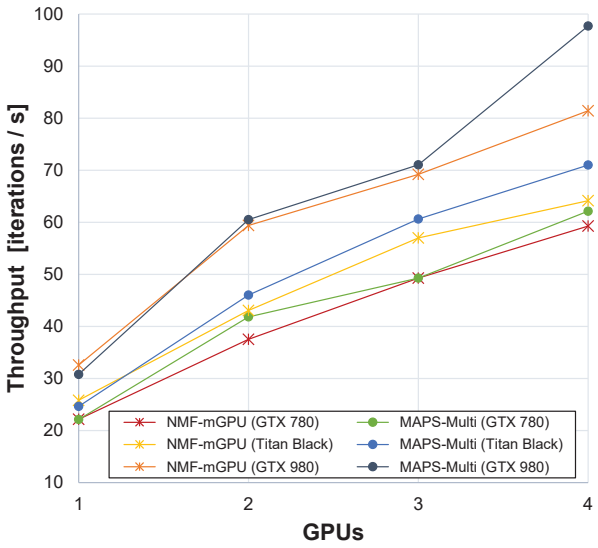


Figure 13: NMF performance

Analysis of the NMF-mGPU source code shows that the GPU kernels are highly optimized for the Kepler architecture, containing ILP optimizations and using specialized instructions. However, its multi-GPU support (on a single node) was implemented over MPI. Consequently, device-to-device memory exchanges pass through the host and are subject to MPI and IPC-related latencies. In contrast, MAPS-Multi uses direct peer-to-peer memory transfers, without involving the host.

7. RELATED WORK

Multi-GPU platforms are attractive for running a multitude of HPC applications [1, 22, 25, 30]. Until recently, achieving adequate performance and scaling required manual tuning of each application to the target architecture and number of devices.

The complexity of programming multi-GPU applications has led to the development of several generic libraries. The SkePU [13] and SkelCL [29] libraries are based on algorithmic skeletons, in which kernels are reduced to one of several types (e.g. Map, MapReduce, MapOverlap) and partitioned accordingly. This resembles the pattern-based partitioning approach in MAPS-Multi, with the exception that the above libraries allow only one output memory access pattern per kernel, impairing development flexibility.

The MGPU [27] library presents a memory partitioning

concept, called segmented containers. These containers are equally divided between the devices, similarly to our *Structured Injective* output access pattern.

Another approach, based on static code analysis and source-to-source compilation, was proposed in [17]. This approach attempts to predict the indices that are used by each device during runtime, in order to partition the tasks. It does not categorize memory access patterns, but rather computes specific accessed indices on the CPU prior to running each kernel. This allows the compiler to predict the exact indices that should be transferred to each device, at the cost of the extra index computation overhead. In contrast, MAPS-Multi uses programming hints and does not require the additional compilation step.

Parallel and distributed algorithms have spawned many programming languages [4, 5, 28] and language extensions [15, 31] that partition the available memory among different nodes, leveraging memory locality for automatic optimizations, while still performing implicit data transfers between nodes. This Partitioned Global Address Space (PGAS) [12] approach is similar to the MAPS-Multi programming paradigm, particularly the use of N-dimensional segmented data structures and the *Window (ND)* input access pattern.

Furthermore, PGAS languages provide common task and data parallelism capabilities, such as atomic operations and parallel reduction, in the form of language keywords and procedures. On the other hand, MAPS-Multi can provide the same capabilities, but requires the programmer to explicitly implement them with respect to the underlying access patterns.

8. CONCLUSIONS

The paper presented a task partitioning and device-level optimization framework. It showed that by specifying the memory access patterns of a kernel, it is possible to automatically distribute the workload among multiple GPUs, as well as perform architecture-specific optimizations.

We showed that the resulting code is intuitive and easy to program. The presented host-level API and device-level iterators hide the underlying index offsets, thread-level optimizations and boundary exchanges.

The performance of memory access pattern-based kernels was tested on three GPU architectures, showing comparable results to production-level single GPU libraries. The framework achieved near-linear scaling on multiple GPUs, reaching speedups of up to $\sim$3.94$\times$ on 4 GPUs for some applications.

The work described in this paper can be extended in several directions. First, the *Unstructured Injective* and *Irregular* output patterns may contain weak forms of ordering that could be leveraged when partitioning tasks. Additionally, the memory analysis phase may be automated via compile-time analysis. It would also be interesting to introduce inter-kernel optimizations, such as concurrent execution and reordering, which require workload estimation for each kernel.

An extension of the MAPS-Multi paradigm to clusters is currently being researched. In distributed HPC environments, communication latency is orders of magnitude higher than within a multi-GPU node. This requires further research into the relation of memory access patterns to device topology and task scheduling, taking network bandwidth considerations into account.

9. ACKNOWLEDGMENTS

This research was supported by the Ministry of Science and Technology, Israel and by the German Research Foundation (DFG) Priority Program 1648 “Software for exascale Computing” (SPP-EXA), research project FFMK. The authors wish to thank the anonymous referees for their constructive comments and suggestions.

10. REFERENCES

[1] M. Ament, G. Knittel, D. Weiskopf, and W. Straßer. A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 583–592, 2010.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[3] J. P. Brunet, P. Tamayo, T. R. Golub, and J. P. Mesirov. Metagenes and molecular pattern discovery using matrix factorization. *Proceedings of the National Academy of Sciences*, 101(12):4164–4169, 2004.

[4] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA ’05, pages 519–538, 2005.

[6] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1337–1345, 2013.

[7] R. Collobert, S. Bengio, and J. Mariéthoz. Torch: a modular machine learning software library. Technical report, IDIAP, 2002.

[8] CUB Library Documentation, 2015. http://nvlabs.github.io/cub/.

[9] CUBLAS Library Documentation, 2015. http://docs.nvidia.com/cuda/cublas/.

[10] NVIDIA cuDNN Deep Learning Library, 2015. http://developer.nvidia.com/cuDNN.

[11] CUFFT Library Documentation, 2015. http://docs.nvidia.com/cuda/cufft/.

[12] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter. Partitioned global address space languages. *ACM Comput. Surv.*, 47(4):62:1–62:27, 2015.

[13] J. Enmyren and C. W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*, HLPP ’10, pages 5–14, 2010.

[14] M. Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

[15] High Performance Fortran language specification. *SIGPLAN Fortran Forum*, 12(4):1–86, 1993.

[16] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678, 2014.

[17] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 277–288, 2011.

[18] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[19] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[20] Y. LeCun and C. Cortes. The MNIST database of handwritten digits, 1998. http://yann.lecun.com/exdb/mnist.

[21] MAPS Framework Documentation, 2015. http://maps-gpu.github.io/.

[22] E. Mejía-Roa, D. Tabas-Madrid, J. Setoain, C. García, F. Tirado, and A. Pascual-Montano. NMF-mGPU: non-negative matrix factorization on multi-GPU systems. *BMC Bioinformatics*, 16(1):43, 2015.

[23] T. Ramashekar and U. Bondhugula. Automatic data allocation and buffer management for multi-GPU machines. *ACM Trans. Archit. Code Optim.*, 10(4):60:1–60:26, 2013.

[24] E. Rubin, E. Levy, A. Barak, and T. Ben-Nun. MAPS: Optimizing massively parallel applications using device-level memory abstraction. *ACM Trans. Archit. Code Optim.*, 11(4):44:1–44:22, 2014.

[25] E. Rustico, G. Bilotta, A. Herault, C. Del Negro, and G. Gallo. Advances in multi-GPU smoothed particle hydrodynamics simulations. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):43–52, 2014.

[26] M. L. Sætra and A. R. Brodtkorb. Shallow water simulations on multiple GPUs. In *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 56–66. Springer, 2012.

[27] S. Schaetz and M. Uecker. A multi-GPU programming library for real-time applications. In *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Part I*, ICA3PP'12, pages 114–128. Springer-Verlag, 2012.

[28] L. Snyder. *Programming Guide to ZPL*. MIT Press, Cambridge, MA, 1999.

[29] M. Steuwer, P. Kegel, and S. Gorlatch. Towards high-level programming of multi-GPU systems using the SkelCL library. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1858–1865, 2012.

[30] J. C. Thibault and I. Senocak. CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, 2009.

[31] UPC Consortium. UPC Language and Library Specifications, v1.3. Technical report, Lawrence Berkeley National Lab, 2013.

[32] C. G. Xanthis, I. E. Venetis, and A. H. Aletras. High performance MRI simulations of motion on multi-GPU systems. *Journal of Cardiovascular Magnetic Resonance*, 16(1):48, 2014.