MAPS: Optimizing Massively Parallel Applications Using Device-Level Memory Abstraction

ERI RUBIN, ELY LEVY, AMNON BARAK, and TAL BEN-NUN,
The Hebrew University of Jerusalem

GPUs play an increasingly important role in high-performance computing. While developing naive code is straightforward, optimizing massively parallel applications requires deep understanding of the underlying architecture. The developer must struggle with complex index calculations and manual memory transfers. This article classifies memory access patterns used in most parallel algorithms, based on Berkeley's Parallel "Dwarfs." It then proposes the MAPS framework, a device-level memory abstraction that facilitates memory access on GPUs, alleviating complex indexing using on-device containers and iterators. This article presents an implementation of MAPS and shows that its performance is comparable to carefully optimized implementations of real-world applications.

Categories and Subject Descriptors: C.1.4 [**Parallel Architectures**]: GPU Memory Abstraction

General Terms: Parallelism, Abstraction, Performance

Additional Key Words and Phrases: GPGPU, memory abstraction, heterogeneous computing architectures, memory access patterns

ACM Reference Format:
Eri Rubin, Ely Levy, Amnon Barak, and Tal Ben-Nun. 2014. MAPS: Optimizing massively parallel applications using device-level memory abstraction. ACM Trans. Architec. Code Optim. 11, 4, Article 44 (December 2014), 22 pages.
DOI: http://dx.doi.org/10.1145/2680544

1. INTRODUCTION

In recent years, the use of Graphic Processing Units (GPUs) has become widespread for real-time and High-Performance Computing (HPC) applications. By efficiently utilizing a large number of cores, GPUs are able to increase the performance of the applications by several orders of magnitude. To achieve such a gain, parallel GPU applications must be carefully optimized, a task that can prove to be quite challenging. This can be attributed to the complex management of the device's multilevel memory architecture. The challenge becomes increasingly daunting with the surge in the number of cores compared to the relatively moderate increase in memory bandwidth and the size of per-core memory.

The memory hierarchy of the GPU typically consists of three layers: (1) persistent global memory, which is relatively large but has high access latency; (2) shared memory,

This research was supported by the Ministry of Science and Technology, Israel.
E. Rubin and E. Levy contributed equally to this article.
Authors' address: E. Rubin, E. Levy, A. Barak, and T. Ben-Nun (Corresponding author), Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel; email: talbn@cs.huji.ac.il.

DOI: http://dx.doi.org/10.1145/2680544

which is faster but restricted to a group of threads; (3) and the per-thread registers, also referred to as private memory. Using shared memory, either as a manually-managed cache or as a local "scratch-pad," can significantly improve the performance of most GPU applications. Such use, though, requires deep knowledge of the algorithm's underlying memory access pattern. Moreover, it compels the developer to manually manage and transfer data between the different levels, taking cache sizes and data ordering into consideration. This management often leads to complex index computations, due to data replication in several layers. This type of programming is complicated and error-prone, requiring expertise in code optimization even for simple applications.

To gain a deeper understanding of memory access patterns in parallel algorithms, [Asanovic et al. 2006] conducted a comprehensive classification of real-world applications and identified 13 common parallel design patterns, called *dwarfs*.

Classic programming libraries, such as the C++ Standard Template Library (STL) [Stroustrup 2000; Musser et al. 2001] and Boost [Boost 2014], provide programmers with abstract tools, based on various design patterns, to access and maintain different types of data structures. Mapping these patterns to parallel systems and GPUs was performed by Hoberock and Bell [2010], CUB [2014], and An et al. [2003]. However, the results are restricted to a small set of predefined algorithms.

Although extensive, the parallel programming dwarfs focus on the communication patterns of distributed algorithms, without addressing the local memory access patterns of the underlying problems. Such a classification would enable the research of parallel algorithms from the memory access perspective. Furthermore, as with the dwarfs, libraries can provide a set of general tools for programmers that optimize common memory access patterns in parallel algorithms.

This article presents the Memory Access Pattern Simplification (MAPS) framework, a device-level abstraction that facilitates memory storage and access on GPUs, while hiding the underlying architecture-dependent read/write optimizations. By investigating the memory access patterns used by the parallel dwarfs, we identified a set of "memory dwarfs," which represent most of the commonly used algorithms for massively parallel architectures. These patterns are then used to derive a general memory abstraction model for massively parallel systems.

The MAPS framework includes three components: parallel iterable containers, device-level algorithms, and index map generation. The first component manages memory transfers between different levels of the GPU memory hierarchy using shared memory objects and thread-level iterators. Device-level algorithms complement the first component by providing efficient algorithms based on these containers; and the third component uses host-side preprocessing to determine access patterns and data overlap regions at runtime. Using these device-level building blocks alleviates the complexity of both index management and efficient memory usage, effectively decoupling the algorithm from its underlying memory access patterns. Unlike current device-level libraries, MAPS grants developers full control over the computational portion of the algorithm, allowing any application to benefit from memory access optimizations.

The article is organized as follows. Section 2 discusses the background for this work. Section 3 introduces the concept of memory dwarfs, and Section 4 extends the concept to the memory abstraction framework. Section 5 demonstrates the performance of a CUDA implementation of MAPS. Section 6 presents related work and Section 7 our conclusions and proposed future work.

2. BACKGROUND

This section describes current GPU architectures, memory optimizations, and parallel programming patterns used throughout the article.

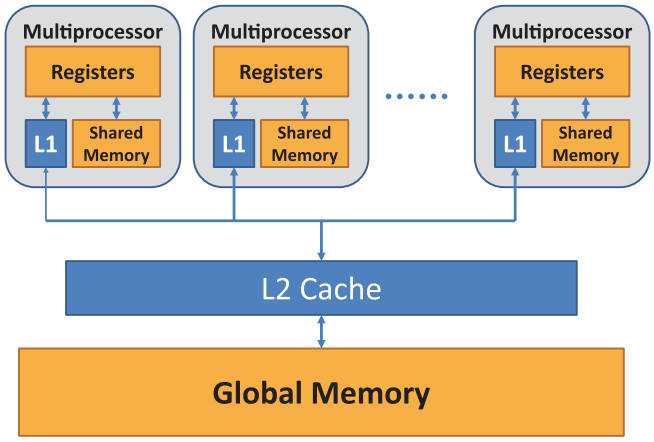

Fig. 1. Detailed GPU memory hierarchy.

2.1. GPU Architecture and Programming Model

State-of-the-art GPU processors consist of a RAM module and several multiprocessors. Each multiprocessor can run multiple *threads*, organized in *blocks* (also referred to as *thread-blocks*). GPUs typically schedule parallel device-level procedures, called *kernels*, to a grid of such blocks. These blocks execute instructions concurrently in lock-steps, that is, threads are divided to groups that execute the same instruction in parallel. Such a group of threads is called a *warp* and has a fixed, architecture-dependent size. If there is a divergence in the execution path of a block (e.g., branching), the divergent threads in the warp become inactive. GPUs attempt concurrent scheduling of as many warps as possible, context-switching between them in order to hide the latency of instructions that require several cycles. Examples for high-latency instructions in GPUs include double-precision and memory I/O operations.

The GPU architecture comprises a multilevel memory hierarchy, as portrayed in Figure 1. The aforementioned RAM module is referred to as the *global memory* and typically contains gigabytes of data. Global memory is used as a bridge between the host and the device, and it has relatively high access latency. Next is the multiprocessor on-chip memory, named *shared memory*. Shared memory is composed of equally sized segments called *banks*, and is accessible to threads in the same block. It has significantly lower access latency than the global memory (approximately 2 orders of magnitude), yet the data stored within it remains only for the runtime of the block. As with other architectures, actual operations are performed on thread-local *registers*, also known as *private memory*.

In addition to the memory types just mentioned, the GPU architecture contains several special-purpose caches. For instance, the texture cache, which acts as a mediator between the global and shared memory, employs a two-dimensional (2D) prefetching mechanism to assist specific types of kernels (e.g., image processing). Some modern GPUs include L1, L2, and L3 caches for storing instructions and data from global memory at lower latencies, improving general performance.

To maximize the performance of memory I/O operations on GPUs, global memory must be accessed in a coalesced fashion, that is, each warp of threads accesses an aligned block of contiguous global memory. When reading or writing to shared memory, each thread (in a warp) should access a different memory bank; otherwise, the operations will be serialized. However, an exception to this occurs when all threads read from the same address—this is optimized in GPUs and called *broadcasting*.

2.2. Shared Memory Optimizations

When programming GPU applications, one of the primary challenges is managing the memory efficiently. The memory access patterns used by the threads can have a dramatic impact on the performance of the application. There are three main conditions where using shared memory is beneficial, referred to in this article as *overlap, order-of-access*, and *scratch-pad*.

The first condition, overlap, occurs when several threads contain overlapping in their memory requirements. In this case, the latency of accessing the global memory can be reduced by performing less reads. An example for this is 2D image convolution, where each pixel requires its immediate neighborhood of pixels for the computations.

The order-of-access condition occurs when threads use input data in a nonsequential order. For example, transposing a matrix requires column-wise reading on matrices that are row-major or vice versa. In this case, coalescing cannot be used to read the data efficiently. Therefore, reading an entire block to the shared memory in a coalesced manner can be much more effective.

The third condition refers to the case where multiple threads share the same block of memory for input/output operations, namely, use a local "scratch-pad." An example for this condition is histogram computation. Histograms are defined by the discrete probability distribution function of given data by assigning its elements to bins. When computing histograms on GPUs, each thread typically processes a single element and increments the bin that corresponds to its value. As the incrementation is an atomic operation, updating the global memory from each thread is highly inefficient. Therefore, traditional histogram implementations that are based on atomic instructions operate on the thread-block level; updating a shared scratch-pad buffer that corresponds to the region assigned to the block. After a thread-block finishes updating the scratch-pad, a single atomic operation is performed to commit the modifications to the global memory, reducing the amount of atomic operations and thus increasing the performance of the algorithm.

If one or more of the aforementioned conditions apply, using shared memory for kernel optimization raises an important issue—index management. While a naive approach to memory access in GPU applications requires some degree of index calculations, optimized memory access often leads to complex indexing. An illustrative example of this issue is shown in Figure 2. The figure shows the programming scheme of an optimized version of 2D image convolution using shared memory, which is comprised of three different representations of the same data. When optimizing this algorithm, there are three subtleties that have to be taken into consideration. First, the data must be transferred in a coalesced fashion from the global memory to the shared memory for efficient access. Second, the shared memory should contain the union of the regions required by threads in the thread-block, and thus contains a larger region with outside boundaries. These boundaries are referred to as the "apron" of the region, and depend on the kernel size of the convolution. Third, the structuring of the shared memory requires the programmer to compute indices with respect to the line-step (or stride) of the shared block, these indices are determined both by the apron size and the thread-block dimensions. Furthermore, the programmer must define strict border conditions for the boundaries of the global input matrix (e.g. mirroring, replication, wrapping). The end result, as shown in the figure, consists of relatively long expressions that are error-prone and hard to understand.

2.3. Parallel Programming "Dwarfs"

To categorize and evaluate parallel algorithms, Asanovic et al. [2006] formulated a list of 13 parallel programming models, called "dwarfs." Each dwarf represents a certain

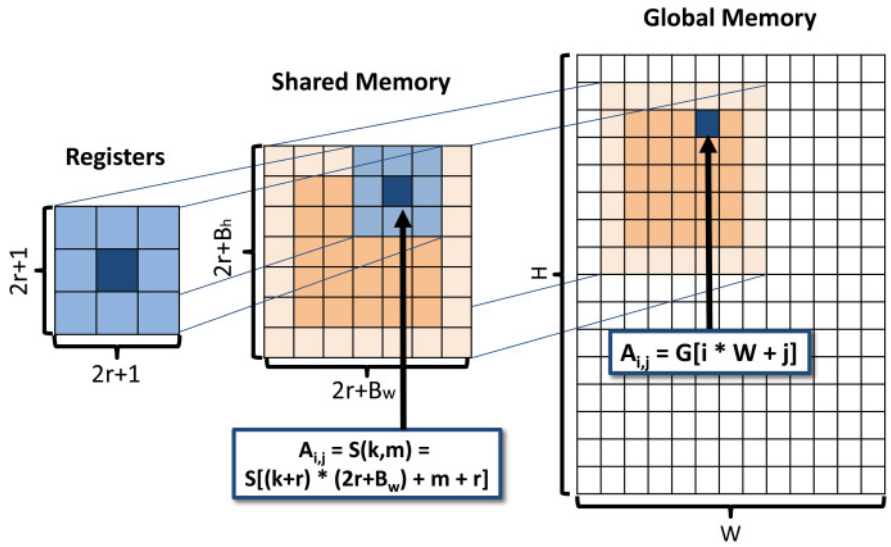

Fig. 2. Programming scheme of a memory-optimized 2D image convolution kernel.

characteristic of parallel applications, and together they cover the vast majority of parallel algorithms. The report provides in-depth analysis as to the communication and computational patterns of these dwarfs, as well as examples ranging from molecular dynamics to machine learning. For reference, a short summary of the dwarfs is given in the following text:

(1) **Dense Linear Algebra**: Performs linear algebra operations, such as multiplication, on dense matrices and vectors.
(2) **Sparse Linear Algebra**: Performs linear algebra operations on sparse matrices, accessing data using indexed memory indirection in order to save memory.
(3) **Spectral Methods**: Represents transformations to and from the frequency domain. This dwarf includes the Cooley-Tukey Fast Fourier Transform (FFT) algorithm and its variants.
(4) **N-Body Methods**: Represents particle-particle force computation methods, where every particle depends on all the others. This dwarf also includes the hierarchical particle methods (e.g., Barnes-Hut Algorithm) and other N-Body methods (e.g., Fast Multipole Method).
(5) **Structured Grids**: Acts on a regular grid, where all the points on the grid are updated during a single iteration by using the values of the neighbors of each point. For example, 2D convolution.
(6) **Unstructured Grids**: This dwarf performs the same update operation as Structured Grids but acts on irregular grids, such as graphs, requiring a level of memory indirection.
(7) **MapReduce**: Typically consists of a single function that executes in full data-parallelism (i.e., embarrassingly parallel). The outputs of this function are combined (e.g. summation) in order to produce a smaller set of results. A representative example of this dwarf is Monte-Carlo Integration.
(8) **Combinational Logic**: Represents operations that are implemented using stateful and logical functions (e.g., computing checksums using Cyclic Redundancy Check (CRC)).

Table I. Memory Access Pattern Categorization by Parallel Dwarfs

Parallel Dwarf	Data Structure	Access Patterns	Typical Example
Dense Linear Algebra	Vector-Vector	Block (1D)	Rank-1 Update ($v \cdot u^T$)
	Matrix-Vector	Block (2D, 1D)	Matrix-Vector Mult.
	Matrix-Matrix	Block (2D, Transposed)	Matrix Mult.
Sparse Linear Algebra	CSR/CSC Matrix	Adjacency	SpMV
	Banded Matrix	Block (1D)	Banded Solver
Spectral Methods	Vector/Matrix	Permutation	FFT
N-Body Methods	Octree	Traversal	Barnes-Hut N-Body
	Array	Block (1D)	$O(n^2)$ N-Body
Structured Grids	Matrix	Window	Convolution
Unstructured Grids	Graph	Adjacency	Cloth Simulation
MapReduce	Varies	Varies	Histogram
Combinational Logic	Varies	Varies	CRC
Graph Traversal	Graph	Traversal/Adjacency	BFS
Dynamic Programming	Varies	Varies	Needleman-Wunsch
Backtrack/Branch-and-Bound	Varies	Varies	A*
Graphical Models	Varies	Varies	HMM
Finite State Machine	Varies	Varies	Any FSM

(9) **Graph Traversal**: Traverses graphs by visiting nodes and following their edges, an example being Breadth-First Search (BFS).
(10) **Dynamic Programming**: Consists of problems that are solved by using the solutions of smaller problems with recursion. These problems are mainly used in optimization (e.g., shortest-path algorithms, longest common string subsequence).
(11) **Backtrack, Branch-and-Bound**: This dwarf represents problems that their solution space can be divided into subspaces, and the optimal solution is obtained by removing the subspaces that produce suboptimal or infeasible solutions. For example, the A* path-finding algorithm.
(12) **Graphical Models**: Builds graphs from probabilistic models. Examples include Hidden Markov Models (HMM) and Bayesian networks.
(13) **Finite State Machine**: This generic dwarf represents problems that are defined by states and conditional transitions that depend on inputs and the current state.

3. CLASSIFYING MEMORY ACCESS PATTERNS IN PARALLEL ALGORITHMS

The categorization of parallel algorithms to programming models, as presented in Section 2.3, is comprehensive both in terms of computation and interprocess communication. However, it lacks an aspect that is very important in modern massively parallel architectures—memory I/O. This section analyzes the common uses of the 13 Parallel Programming Dwarfs with respect to their various memory access patterns, noting typical examples implemented efficiently on the GPU architecture.

Table I enumerates the different parallel dwarfs, the respective data structures that can be used in each one, and classifies input/output access patterns with respect to massively parallel architectures. As shown in the table, data structures are reduced to basic Linear Algebra (LA) primitives (vectors and matrices) alongside graph-based structures (graphs and trees). The access patterns shown in the table address both input and output memory operations, with the example of 2D and 1D blocks in Matrix-Vector Dense LA, representing the matrix and the vector in the operation, respectively. Additionally, some dwarfs, such as Dynamic Programming and Finite State Machines (FSM), are too generic to find recurring data structures and memory access patterns.

Generalizing the above table to massively parallel architectures, we obtain a list of seven *"Memory Dwarfs"*:

(1) **Block (1D)**: Continuous memory that can be loaded to thread-blocks in chunks, due to data reuse in several threads and the benefits of reading sequential data. This access pattern can be used for accessing vectors and banded matrices. Examples include vector access in dense matrix-vector multiplication and particle array access in the all-pairs $O(n^2)$ N-Body solver [Nyland et al. 2007].
(2) **Block (2D)**: Continuous multirow memory that can be loaded to thread-blocks in horizontal tiles. The difference between this and **Block (1D)** is the efficient addressing of 2D kernels while maintaining coalesced global memory access. A typical usage example is reading the first matrix in dense matrix multiplication.
(3) **Block (2D-Transposed)**: Multicolumn memory that can be loaded to thread-blocks in vertical tiles. When using shared memory for this case, the performance benefits are significant, as the data can be read and copied in row-major blocks and then accessed from the shared memory in a column-wise manner, thus eliminating all noncoalesced memory reads. This memory access pattern is used both when reading matrices for matrix transposition and when reading the second matrix in dense matrix multiplication.
(4) **Window (1D, 2D, 3D)**: Represents data that is not continuous in memory but is spatially local. The underlying algorithm should contain a certain degree of overlapping in the per-thread data requirements. For efficient access, data has to be read in regions covering an entire thread-block's requirements. In theory, this pattern applies to any N-dimensional window; however, most algorithms use 1D to 3D windows. Examples include 2D image convolution, where the neighborhood of each pixel has to be loaded in order to compute the resulting image; and any algorithm based on the Structured Grid dwarf.
(5) **Adjacency**: This access pattern is suitable for sparse matrix operations and graphs. These data structures cannot be stored as contiguous memory, and thus algorithms often access data sporadically. Because the data does not have a predetermined structure, preprocessing is necessary to determine which data should be loaded to the shared memory for each thread-block. This pattern is mostly used for efficient Sparse Matrix-Vector multiplication (SpMV) [Garland 2008] and algorithms in the Unstructured Grids dwarf.
(6) **Traversal (DFS, BFS)**: Represents tree and graph data structures, where each thread-block is required to operate on neighbors of a certain node. An example for this memory access pattern is the Barnes-Hut Algorithm [Burtscher and Pingali 2011], a hierarchical approximation of the N-Body problem.
(7) **Permutation**: Primarily used in the Spectral Methods dwarf, this memory access pattern loads a contiguous block of information and distributes it to the thread-block in a specific permutation. After performing the necessary computations, the data is stored in another permutation. If global memory access is performed naively, it may have a major impact on the performance of the algorithm. In standard GPU implementations of FFT, several stages of permutations, called "butterfly" stages, combine the results of smaller Discrete Fourier Transforms to the final result.

Not all parallel dwarfs can be injectively mapped to the aforementioned list of memory dwarfs. Some parallel dwarfs do not have a direct memory dwarf counterpart, examples being Graphical Models and Backtrack/Branch-and-Bound. These dwarfs can benefit from massively parallel architectures by simply assigning different initial estimates per-thread. Additionally, Finite State Machines are sequential by design and thus cannot be directly mapped to any memory dwarf. Other parallel dwarfs, such as MapReduce and Dynamic Programming, vary in data structures (e.g., vectors, matrices, tensors) and access patterns (e.g., histograms, word counting, sorting). Finally, some algorithms can be represented by more than one memory dwarf. For example,

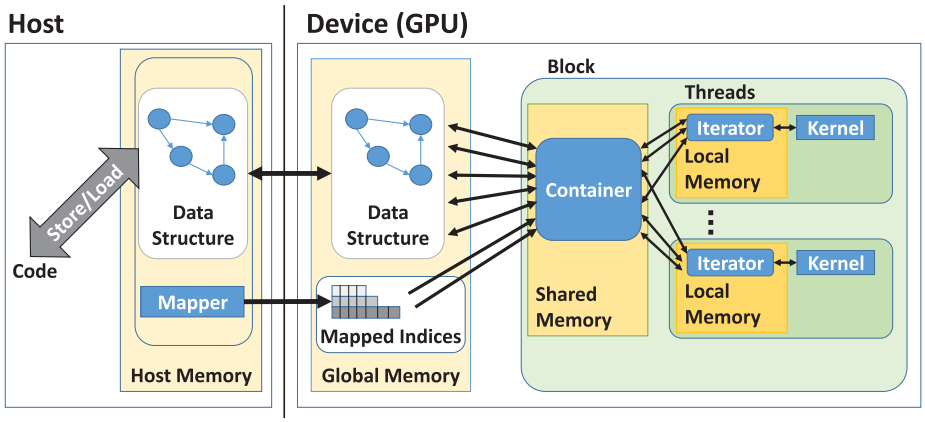

Fig. 3. Suggested device-level memory abstraction.

Breadth-First Search (BFS) can be implemented both by the **Adjacency** and **Traversal (BFS)** memory dwarfs. This is because BFS can be formulated as the result of consecutive SpMV operations, where the $\times$ and $+$ operators are replaced with $+$ and *min* respectively [Hong et al. 2011].

4. MAPS: DEVICE-LEVEL MEMORY ABSTRACTION

After classifying the prevalent memory access patterns in massively parallel architectures, we proceed to present a memory abstraction that alleviates the so-called *indexing hell* that often accompanies memory optimizations, while maintaining optimized input and output for each of those patterns. There are three components to this abstraction: Parallel iterable containers, device-level algorithms, and index map generation. The first component facilitates memory access on the device, while hiding the underlying architecture-dependent read/write optimizations; the second component complements the first by providing generic block-wide algorithms that utilize the containers to efficiently access data; and the third component preprocesses inputs on the host to determine access patterns and data overlap regions at runtime. Based on the extensiveness of the 13 dwarfs model, most parallel algorithms can be reduced to one or more memory dwarfs. Therefore, we can separate massively parallel algorithms to four building blocks: input containers, thread-level core operations, device-level algorithms, and data output.

Figure 3 presents the aforementioned components of the suggested device-level memory abstraction. From the figure, it can be seen that host code (on the left side of the diagram) does not change considerably, whereas device code (on the right side) accesses memory via iterators and containers, rather than accessing global and shared memory directly.

In the rest of this section, we describe the three components of MAPS in detail, followed by example usage of the framework on GPU matrix multiplication.

4.1. Parallel Iterable Containers

Parallel containers are device-level classes that manage access to data structures in specific access patterns. Parallel containers bear some resemblance to the commonly used STL containers; however, there are a few fundamental differences. Specifically, containers operate on the thread-block level: each block contains an instance of the container in shared memory, while the main copy of the data resides in the global memory. As each block requires a specific portion of the data, the block container will

Table II. Parallel Container Device-Level Programming Interface

device class Container	Method Description
init(...)	Initializes the container with a specific data structure.
begin()	Creates a thread-level iterator that points to the beginning of the current chunk.
end()	Creates a thread-level iterator that points to the end of the current chunk.
nextChunk()	Progresses to process the next chunk.
isDone()	Returns true if all chunks have been processed.
device class Container::Iterator	**Method Description**
operator++()	Advances the iterator.
operator*()	Obtains the value of the current iterator location.

fetch only the required part. These parts can be noncontinuous and often overlap with the ones needed by other blocks. From a database management standpoint, a container creates a "local view" of the data, which automatically calculates the indices in order to load the necessary data to the block. In case the required data is larger than the amount of shared memory, it is loaded and processed in separate chunks.

Because parallel containers are implemented as classes, they can act as a storage object that provides both an inheritable method-based interface, and internal classes that iterate over the contained data. The internal classes that are instantiated from the parallel containers are called **thread-level iterators**. These objects are only visible to their instantiating threads, and, in contrast to STL iterators, only possess a thread-local view of the parallel container. Usually, there is a certain degree of overlapping between the data accessed by different thread-level iterators. The data may also not be consecutively addressed or even contiguous in the actual data structure, as iterators transparently manage addressing. Note that multiple iterators can be spawned on a single thread (e.g., with different starting points) to increase Instruction-Level Parallelism (ILP).

Table II presents the base APIs of the parallel containers and thread-level iterators proposed in this article. The data pointed to by the container is transferred from the global memory in chunks and on request, using the `nextChunk` method. To access the actual data, generic indexless iterators are created on threads using the `begin` and `end` methods. The iterators access and advance the data pointer using the `*` and `++` operators, respectively.

4.2. Device-level Algorithms

Similarly to STL's container-based algorithms, the proposed device-level memory abstraction includes **device-level algorithms**. These algorithms operate on data obtained from parallel containers, compensating for the absence of data structures that handle certain aspects of GPU development and memory management. Viewing the world from a parallel programming dwarf perspective, these algorithms mainly target the reduction portion of the MapReduce dwarf, which has no specific memory access pattern that can be represented as a parallel container.

Because of their diversity, device-level algorithms do not support a generic programming interface. However, as with parallel containers, all algorithms are represented as classes residing in the shared memory. This is necessary because the algorithms may have additional memory requirements; an example being device-level histograms, which require a shared storage of local bins.

For instance, computing a device-level histogram is implemented as follows. Initialization is performed by `DevHistogram::init(g_bins)`, where `g_bins` is a buffer residing

Table III. Index Mapper Programming Interface

class IndexMapper	Method Description
init(...)	Initializes data structure parameters.
registerData(...)	Registers the data structure for this container.
createIndexMap(...)	Prepares the index map and returns it.
releaseIndexMap(...)	Frees the memory of the allocated index map.

in global memory that will be updated with the results of the device-level algorithm. Block-level processing is performed by calling `DevHistogram::compute(value)`. While all algorithms allow kernels to access the block-level output, global reduction-based algorithms provide an API to safely update global memory, using the `commit` method. Continuing the device histogram analogy, calling `DevHistogram::commit()` finalizes the results on the global memory.

The efficiency of the proposed algorithms derives from multiple architecture-specific implementations, which are often hard for GPU programmers to develop and maintain. Thus, by providing carefully optimized implementations in a separate library, crucial time in GPU algorithm development can be saved. When implementing such algorithms, performance depends on many parameters. Available shared memory, cache size, block size, warp size, low-level instructions and device compute capability are just some of the parameters that should be taken into account. Utilizing these architecture-specific optimizations correctly, when possible, can dramatically increase the performance of kernels.

4.3. Host Index Mapping

Some of the memory dwarfs do not exhibit data locality, that is, there is no efficient way to directly assign tasks to thread-blocks such that the threads reuse the same memory. This may result in sporadic memory access, which significantly hinders performance. However, in some cases, the access patterns and overlap regions can be determined during the runtime of the application. For these cases, the MAPS framework includes the **index mapper** component. This component processes data structures on the host to find an optimal caching strategy for each thread-block. Each parallel container that requires this processing includes its own index mapper implementation, with respect to the underlying data structure. The index mapper generates a data structure, called "index map," which is utilized by the parallel containers on the device. It is then used to load the data efficiently and create thread-level iterators for each thread-block.

This type of processing is computationally expensive; hence, it is only beneficial if three conditions are met: (1) data locality can be achieved, (2) the topology of the data (e.g., the sparsity pattern of a matrix) does not change significantly during the runtime of the application, and (3) the container is used in multiple kernel calls. When these conditions apply, index mapping can be used as a runtime preprocessing stage.

Table III presents the API for index mapping. To use the index mapper, the parameters of the data structure must be specified using the `init` method. Then, the elements are registered to the container using the `registerData` method. To generate a new index map for the registered data, `createIndexMap` is called, which returns the appropriate arrays as device memory, which can then be used by kernels when necessary. If a recomputation of the index map is required (e.g., due to a change of topology in a graph), the two methods just described should be called again.

An example for index mapping is the parallel container based on the **Adjacency** memory dwarf. In this memory dwarf, the underlying data structure is a graph, represented as an adjacency list or a sparse matrix. In either case, efficient memory access can be difficult to achieve, as the connectivity pattern is usually irregular and cannot

```
uint2 *edges = ...;
maps::GraphMapper mapper;

mapper.init(elementsPerBlock, numNodes);

// ...
// Initialize graph data
// ...

mapper.registerData(edges, numEdges);

// ...

// Global -> Shared
uint2 *devBlockOffsetSize;     // Offset and size for each block
uint  *devBlockGlobalIndices;  // List of indices needed by each block

// Shared -> Registers
uint2 *devThreadOffsetSize;    // Offset and size for each thread
uint  *devThreadSharedIndices; // List of indices needed by each thread

// Creates index map and copies it to the device
mapper.createIndexMap(devBlockOffsetSize,  devBlockGlobalIndices,
                      devThreadOffsetSize, devThreadSharedIndices);

// ...
// Run kernel
// ...

mapper.releaseIndexMap(devBlockOffsetSize,  devBlockGlobalIndices,
                       devThreadOffsetSize, devThreadSharedIndices);
```

Fig. 4. Host-code snippet of index mapping for the Adjacency memory dwarf.

be predetermined. However, in many cases the topology of graph does not change (or changes very little) during the runtime of the algorithm.

Figure 4 shows an example usage of the index mapper API for the **Adjacency** memory dwarf. In the example, the index mapper analyzes the connectivity of the graph and builds index arrays for use in the **Adjacency** parallel container. These arrays enable efficient global memory access, while minimizing shared memory bank conflicts. During the runtime of the kernel, the dense graph information in the global memory is copied to block-level segments in shared memory, which are then used by individual threads. As shown in the figure, the graph index map is internally described by four arrays: `devBlockGlobalIndices`, `devBlockOffsetSize`, `devThreadSharedIndices`, and `devThreadOffsetSize`. The first array contains the block-level list of data indices that have to be loaded from the global memory to the shared memory. The second array maintains offsets and sizes for each block to access the first array. The third array contains a list of shared memory indices that are needed to be loaded by each thread, and the fourth array contains the offsets and sizes for accessing the third array. These arrays are all organized for optimized access using coalescing and padding. Note that the values of the arrays depend only on the topology of the graph, thus they can be used regardless of the values of its nodes.

4.4. Usage Example

To show the simplicity of the MAPS framework, we present a GPU implementation of dense matrix multiplication using parallel containers based on the **Block2D** and **Block2D-Transposed** memory dwarfs.

Naive

```
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;
int aBegin = n * BW * by;
int bBegin = BW * bx;



for (int i = 0; i < n; ++i)



{
    Csub += A[aBegin + n * ty + i]
          * B[bBegin + k * i + tx];
}
```

MAPS

```
__shared__ Block2D <float,BW> matConA;
__shared__ Block2DT<float,BW> matConB;
matConA.init(A, m, n);
matConB.init(B, n, k);
Block2D <float,BW>::iterator matAIt;
Block2DT<float,BW>::iterator matBIt;

while (!matConA.isDone())
{
    for (matAIt = matConA.begin(),
         matBIt = matConB.begin();
         matAIt != matConA.end();
         ++matAIt, ++matBIt)
    {
        Csub += (*matAIt)
              * (*matBIt);
    }

    matConA.nextChunk();
    matConB.nextChunk();
}
```

Fig. 5. Naive versus MAPS device-code snippets of matrix multiplication.

Figure 5 compares a code snippet of a naive implementation (left column) to the corresponding snippet using MAPS, both implemented in CUDA. The input matrices A and B are of sizes $m \times n$ and $n \times k$, respectively, and the number of threads per block is BW. As shown in the figure, even the naive implementation requires some degree of indexing, which is done to compute different portions of the output matrix on each thread. In contrast to the naive implementation, the MAPS-based code does not contain any explicit indices. Instead, it utilizes two iterators, operating on chunks of the input matrices and hiding the complexity of memory transfers and shared storage from the programmer. Also note that the readability of the code is not compromised by the framework, but rather emphasized. As for performance, Section 5.3 shows that the MAPS version performs up to $\sim$2.64$\times$ faster than the naive version.

5. PERFORMANCE EVALUATION

In this section, we present the performance of the CUDA implementation of our framework through a set of example applications, representing various parallel dwarfs corresponding to different memory access patterns. Each example compares the framework-based implementation with both naive and carefully optimized implementations. The naive implementations are simple, straightforward, and do not utilize any memory optimizations. The carefully optimized versions, on the other hand, use complex application-specific memory optimizations meant to utilize the full capabilities of the device. We should note that the source code of the optimized version is much longer and more complex than the other two versions.

Our experimental setup consists of an NVIDIA Tesla K40c GPU, containing 12GB of global memory and 48KB of shared memory per block; and an NVIDIA GeForce GTX 750 Ti GPU, with 2GB of global memory and 64KB of shared memory per block. These devices are based on the Kepler and Maxwell architectures, respectively, where the latter is more advanced and better suited for shared memory operations. In the following graphs, the Tesla K40c is used as the default test platform, and Section 5.8 compares the performance of the two architectures for each of the examples.

5.1. CUDA Implementation

To investigate the efficiency of device-level memory abstraction, we implemented the MAPS framework using the CUDA software development kit [CUDA 2014]. Implementation in CUDA was chosen for two main reasons: first, it is the only device-level language that supports the necessary class and template C++ interfaces; and second, using CUDA allows the framework to be used as a basis for implementation in higher-level languages, such as OpenACC [2012] and C++ AMP [Gregory and Miller 2012]. The code, along with further technical documentation, is available at MAPS [2014].

The current CUDA version of MAPS includes most of the memory dwarfs, implemented as parallel containers. The two exceptions are the **Permutation** and **Traversal** memory dwarfs. The **Permutation** memory dwarf is almost exclusively used for FFT, of which there are highly optimized architecture-specific implementations (such as CUFFT [2014]). Therefore, it will not benefit from a generic container implementation. On the other hand, the **Traversal** memory dwarf cannot be efficiently represented by device-level parallel containers, as it requires modifying the entire programming model of the device code. In particular, it requires bypassing the thread-scheduling model of CUDA in order to manually allocate jobs to threads, similarly to traditional parallel programming [Aila and Laine 2009; Gupta et al. 2012].

The **Block1D** container handles computations that require each thread to process an entire 1D array, or a segment of it. This container reduces the number of global memory reads by loading batches of data to the thread-block shared memory.

Block2D and **Block2D-Transposed** are parallel containers intended for dense matrix operations. These two containers produce thread-level iterators that go over rows and columns of a matrix, respectively. Hence, they can be used for many algorithms in computer graphics and scientific computing. A naive implementation of such algorithms is known to have an intense memory access pattern, as each part of the data has to be read multiple times. By transparently transferring blocks to the shared memory, global memory reads are ensured to be both coalesced and minimal.

The **Window** container represents data that is accessed in a spatially local manner, where each thread is required to process a specific region (or "window") that overlaps with regions required by neighboring threads (i.e., residing in the same block). The implementation of this container loads the union of the required regions (including boundaries), while handling out-of-bound memory accesses using one of the following methods: mirroring, replication, or wrapping.

Another implemented parallel container is **Adjacency**, intended for efficiently processing graphs and sparse matrices. This container also utilizes host **index mapping** to produce data-localized representations of graphs using their topology. To perform well, the adjacency container requires the graph matrix to be index local, that is, elements that are spatially close have numerically close indices. If the graphs are not index local, preconditioning methods such as those by Hoppe [1999] and Sander et al. [2007] can be used to reorder them.

To achieve shared memory coherency throughout the implementation of MAPS, block-level thread synchronization is used in the form of barriers. The synchronizations are typically performed on two occasions: after shared memory initialization and before loading new chunks to the shared memory (using the `nextChunk` method).

All the aforementioned parallel containers are implemented as shared-memory template classes with inline methods. The containers require several parameters to be initialized: Global dimensions of the underlying data structure, for example, matrix size; thread-block dimensions; and specific dwarf-based parameters, such as the shared window dimensions in the **Window** container. The parameters that are known in compile

time are input as template arguments, whereas parameters determined in runtime are passed to the `init` method of the container. Further details regarding implementation specifics are available in the code documentation.

5.2. Performance Optimization

When implementing a framework such as MAPS, two major aspects have to be taken into account: function design and architecture-specific development. There are several tradeoffs for each aspect, which are explained in this section.

The first aspect refers to the choice of design patterns used in the device-code library. This requires careful planning, as performance can be impeded by inefficient constructs, such as virtual functions and polymorphism. The tradeoff between using class-based objects as opposed to static inline functions determines how much performance is sacrificed in favor of generality and flexibility. Preliminary tests showed that the performance increase of inline functions was negligible at best, and thus it was decided to use shared memory classes with inline methods.

As for the second aspect, GPU architectures dramatically differ in performance of memory access operations. An example of architecture variation is in atomic operations. Newer architectures provide better support for atomic operations, and thus should be used whenever possible. Moreover, the Maxwell architecture includes better support for shared memory atomic operations, whereas the Kepler architecture is faster with global memory atomic operations. Therefore, in the device-level histogram algorithm, Kepler-based devices use global atomics and Maxwell-based devices use shared atomics.

The optimizations used in the MAPS implementation can be divided into five groups:

(1) **Coalesced memory reading**: For a memory read to be coalesced, threads in a warp are required to access a contiguous chunk of memory starting from an aligned address. This can be achieved using various methods. For example, using aligned `float4` arrays instead of `float3`, or padding the end of a row in 2D data structures. In the **Adjacency**-based container, the data generated by the index mapper is automatically padded, to allow better coalescing. Generally, this optimization is used in our implementation whenever possible.
(2) **Bank conflict avoidance**: The shared memory is composed of memory banks. As mentioned in Section 2, access operations from more than one thread in a warp to different addresses in the same bank are serialized. This can be avoided by carefully spreading the data across the memory banks. For example, in the **Block2D-Transposed** container, the data is stored using the diagonal block reordering optimization technique [Ruetsch and Micikevicius 2009], which guarantees zero bank conflicts in both read and write operations. Similarly, in the **Window**-based parallel container, rectangular shared windows are used instead of square windows.
(3) **Vectorized data reordering**: To further minimize bank conflicts, whenever vectorized data (e.g., `float3`, `int2`) is stored in shared memory, the data is organized by component and padded according to the bank size for alignment. For example, storing n elements of (x, y, z) coordinates would be organized as $(x_1, \ldots, x_n, y_1, \ldots, y_n, z_1, \ldots, z_n)$.
(4) **Texture cache usage**: Instead of directly reading from global memory, texture caches are used for spatially local reads in two and three dimensions. This optimization applies to the **Window (2D)** and **Window (3D)**-based containers.
(5) **Partition camping avoidance**: When multiple thread-blocks read from the same region of the global memory, it can cause a phenomenon called "Partition Camping." In **Block1D**, this issue is solved by starting the processing of each thread-block

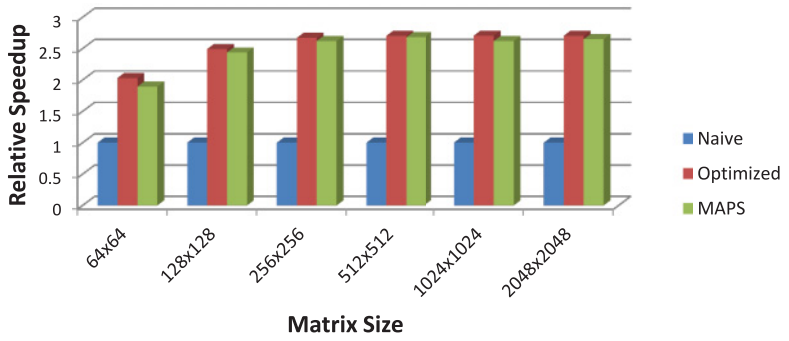

Fig. 6. Relative performance of dense matrix multiplication.

in a different segment of the global memory, wrapping around past the end of the global buffer to reach all segments.

5.3. Dense Linear Algebra: Matrix Multiplication

The first application we show is dense matrix-matrix multiplication, which is an essential base for a plethora of algorithms. The code, which is shown in Figure 5, consists of two input containers: **Block2D** and **Block2D-Transposed**, one for each input matrix. Although there are highly optimized libraries available for this operation [CUBLAS 2014], our framework provides more flexibility, which is especially useful for unconventional cases. Examples include replacing the $\times$ and $+$ operators, adding additional operations to increase instruction-level parallelism, or any other functionalities that are unsupported by the current high-level libraries.

Figure 6 shows the performance of the implementation of this application, and compares it with naive and manually optimized implementations. The naive implementation was written by the authors, whereas the optimized implementation was taken from the CUDA SDK. In the figure, we can see that the factor of the speedup increases with the size of the matrix. This is probably due to the heavy usage of the parallel container interface, which is causing a slight overhead. However, in larger matrices, the performance of using the MAPS framework is approximately 2.6$\times$ faster, compared to the naive implementation, having a negligible overhead compared to the manually optimized version.

5.4. Structured Grids: 2D Image Convolution

The second application shown in this section is 2D image convolution. This example, which is based on the **Window2D** memory dwarf, operates on each pixel to compute a weighted average of itself and its immediate neighbors. The weights for the average are determined by another matrix, called the *convolution kernel*. 2D convolution is a core image and signal processing operation, and has many uses in these areas.

Figure 7 shows the results of performing 2D convolution on a 2,048$\times$2,048 image, comparing the MAPS implementation to the naive and manually optimized ones on various convolution kernel sizes. In the figure, it can be seen that the naive implementation of the 3$\times$3 convolution is faster than both other implementations. This can be attributed to the fact that only 9 pixels are loaded to the shared memory, causing the code to spend more time on thread synchronization than reading from global memory. Additionally, the L1 and L2 caches speed up global memory access automatically for this case. For larger kernels, we can see that the shared memory implementations perform up to $\sim$1.72$\times$ faster than the naive implementation.

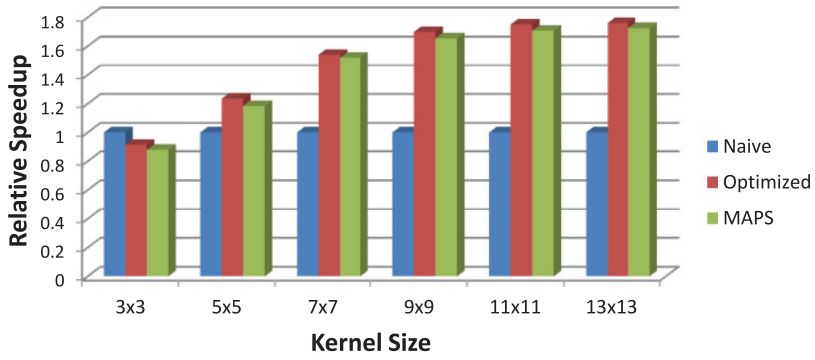

Fig. 7. Relative performance of 2D image convolution.

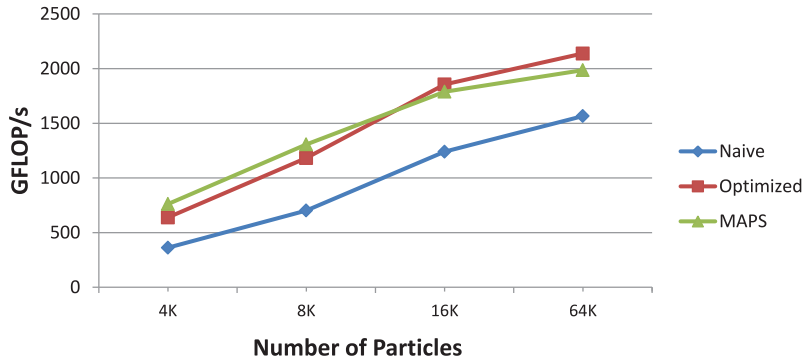

Fig. 8. Throughput graph of the all-pairs N-body simulation implementations.

5.5. N-body Methods: All-pairs Algorithm

This example shows the performance of the $O(n^2)$ all-pairs N-body simulation [Nyland et al. 2007], based on the **Block1D** container. The N-body problem consists of N particles, which apply forces on each other (e.g., gravitational pull of stars). In each step, every particle updates its own location based on its distance from all the other particles. These simulations are widely used in astrophysics and molecular dynamics.

Figure 8 presents the throughput of the three implementations, measured in particle interactions per second. Both the naive and optimized implementations we use are taken from the CUDA SDK N-body application. Because the performance depends on the data size quadratically, it is more informative to depict the throughput, which better shows the difference in performance between the versions. As shown in the figure, for small numbers of particles the MAPS implementation outperforms the optimized implementation. This is probably due to the heavy reliance of the optimized implementation on shared memory classes, which are not utilized as efficiently as in the MAPS implementation.

5.6. Unstructured Grids: Cloth Simulation

This example shows the performance of 3D cloth simulation based on mass-spring systems [Provot 1995], using the **Adjacency** memory dwarf. In this application, the cloth is represented by a polygonal mesh composed of triangles, to which forces are applied based on the mass-spring physical model. In each timestep, the simulation

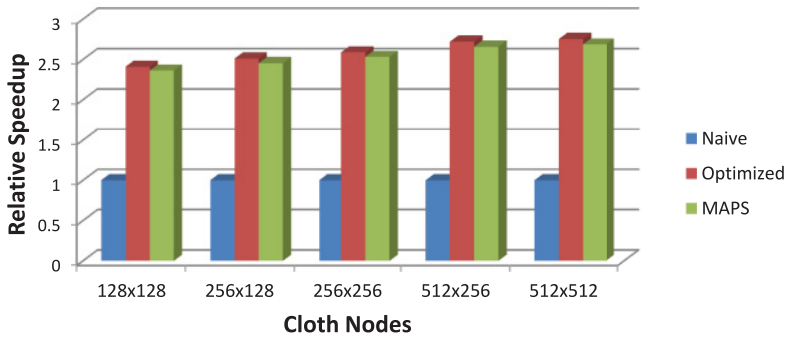

Fig. 9. Relative performance of cloth simulation based on mass spring systems.

updates the forces on the nodes and calculates their new location. Cloth simulation is widely used in physical simulations and computer graphics.

Figure 9 compares the index mapper-based MAPS implementation with naive and manually optimized implementations for different numbers of cloth nodes. It should be noted that the naive version accesses the memory directly, whereas the manually optimized version employs its own index mapping implementation to enable the use of shared memory. In the figure, we see that the overhead of using MAPS for this case does not increase with mesh density, and that it is negligible compared to the $\sim 2.7\times$ speedup over the naive implementation.

It is worth noting that the current implementation of the cloth simulation application does not handle incremental graph topology changes. This means that any change in the graph topology requires complete reprocessing of the index mapper. This is because of the generic implementation of the index mapper in the **Adjacency** memory dwarf. To overcome this limitation, a problem-specific index mapper has to be implemented for this application, handling such cases. For instance, a host process can gradually update the index array maps in parallel to GPU processing.

5.7. MapReduce: Fused Convolution-Histogram

This example shows the usage and performance of device-level algorithms in conjunction with parallel containers, to implement a MapReduce-based algorithm. The algorithm first performs a 2D image convolution (using the same code as in Section 5.4), and then computes the histogram of the convolved image. The main advantage of the fused operation is that it performs both actions on the same kernel, reducing overhead and eliminating the output of unnecessary intermediate results to the global device memory.

The histogram device-level algorithm is also an example for the performance portability of MAPS. Because the two architectures in our experimental setup differ in performance of atomic operations, the internal implementation in MAPS is optimized for each platform. It should also be noted that MAPS automatically chooses the fastest implementation according to the hardware architecture being used.

Figure 10 presents the performance of the fused image convolution and histogram computation, and compares it with the commonly used GPU implementations of histogram computation from CUB [2014] and NPP [2014]. Because of the large performance differences between the two tested devices, the figure shows speedup on both architectures.

The figure shows that the naive implementation performs better than its NPP counterpart (`nppiHistogramEven_8u_C1R`) on both the Kepler-based Tesla K40c and the Maxwell-based GeForce 750 Ti. For the Kepler-based GPU, the fused MAPS version

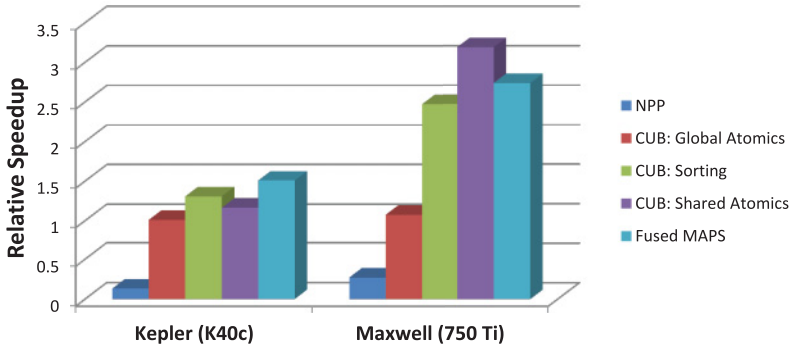

Fig. 10. Relative performance of the convolution-histogram fused operation.

Table IV. MAPS Relative Speedup Summary

Application	GPU Architecture	Optimized Speedup	MAPS Speedup	Optimized/MAPS Speedup
Matrix Multiplication	Kepler	2.551	2.482	0.971
	Maxwell	1.929	1.744	0.905
Cloth Simulation	Kepler	2.596	2.538	0.977
	Maxwell	1.926	1.721	0.894
2D Image Convolution	Kepler	1.483	1.445	0.972
	Maxwell	1.637	1.594	0.969
All-Pairs N-Body	Kepler	1.493	1.498	1.048
	Maxwell	1.605	1.604	1.003
Fused Convolution-Histogram	Kepler	1.307	1.508	1.153
	Maxwell	3.189	2.736	0.858

is the fastest implementation. As mentioned earlier, the Maxwell architecture contains improved shared memory atomic operation support, resulting in the CUB shared atomics being the fastest with the fused MAPS as a close second.

The internal implementations of CUB and MAPS differ substantially, but the performance advantage of CUB can most likely be accounted to its use of Instruction-Level Parallelism (ILP) optimizations, which are done by interleaving independent instructions in the device code. One of the ways to achieve ILP is to increase the work each thread performs by redistributing the processing and launching less threads. This type of optimizations cannot be done by MAPS or compilers automatically, though MAPS allows programmers to implement such optimizations manually. Since CUB consists of a high-level interface, such optimizations are implemented internally, at the cost of programming flexibility. These optimizations demonstrate the tradeoff between producing fast code, as opposed to portable code that is easier to maintain.

5.8. Cross-Architecture Performance

The following measurements summarize the speedup of the aforementioned applications on both the Kepler and Maxwell architectures. Although some of the optimized implementations (cloth simulation and fused convolution-histogram) differ between the architectures, the MAPS-based code is identical on both platforms.

Table IV presents the results of the aforementioned measurements. The first column shows the speedup of the manually optimized implementation relatively to the naive version; the middle column shows the relative speedup of the MAPS implementation

over the naive version; and the right column shows the relative performance of MAPS with respect to the manually optimized version.

The results show that all examples on both architectures achieve speedups between 1.44× and 2.74× over the naive versions. Additionally, despite running on different architectures that vary in memory access speeds and caching methods, the relative performance of MAPS over the optimized version remains similar, ranging between 0.89× and 1.15×. The only exception is the performance of the fused convolution-histogram application, which heavily depends on operations that were considerably optimized in the Maxwell architecture.

6. RELATED WORK

Efficient programming for massively parallel architectures is known to be difficult. Many attempts have been made to ease this task by creating high-level programming models. The first approach is based on programming directives that serve as hints to compilers, examples being OpenACC [2012] and OpenMPC [Lee and Eigenmann 2010]. The second approach, used by C++ AMP [Gregory and Miller 2012] and hiCUDA [Han and Abdelrahman 2011], is purely based on compilers that provide high-level APIs. The third approach, as with MAPS, consists of runtime libraries that provide high-level and performance portable functionality. Notable examples for this approach are Thrust [Hoberock and Bell 2010] and CUB [2014].

The first two approaches significantly reduce the amount of effort required in order to produce moderately optimized code, but when programmers wish to achieve a high level of optimization, these languages are very restrictive and still require manual memory and indexing management. Both libraries in the third approach augment the existing runtime environment by adding customizable kernels (using C++ functors); but while Thrust provides a high-level, STL-like API, CUB provides both a device-level API for calling kernels from host code and a block-level API for calling block-wide methods inside existing kernels. This provides the programmer with a wider set of tools and is similar to the Device-Level Algorithms in the MAPS framework. However, most complex algorithms require additional kernels, which are not implemented by any of the libraries and have to be manually developed. MAPS provides a different perspective to the runtime library approach, and thus complements the works mentioned earlier.

When designing the device-level memory abstraction, we were inspired by the original C++ Standard Template Library (STL) [Stroustrup 2000; Musser et al. 2001] and parallel versions that use its principles, such as STAPL [An et al. 2003] and Glift [Lefohn et al. 2006]. The idea is to use familiar design patterns, containers and iterators, to avoid code complication that often accompanies manual device-code optimization, without limiting the user in any way. Similar device-level libraries like CUDPP [Harris et al. 2007] and CUB mainly provide algorithms, whereas the MAPS framework focuses on alleviating complex indexing created by memory optimizations.

The work done by CudaDMA [Bauer et al. 2011] is similar to MAPS in the sense that it uses a device-level API based on generic classes to hide the complexity of optimizations. However, the MAPS framework uses parallel containers and device-level iterators to describe different memory access patterns, whereas the focus of the device-level API in CudaDMA is on memory-transfer based optimizations, using Direct Memory Access (DMA) and specializing warps when possible. These optimizations can be used in conjunction with MAPS to further improve performance of GPU applications.

As for indexing and device-level memory optimizations, Dymaxion [Che et al. 2011] is a library that simplifies memory management of certain data structures by rearranging input data on the host, prior to running kernels, to ensure coalescing of all data reads on the device. The authors also address the concept of memory access patterns on matrices (i.e. row, column and diagonal) and the appropriate index re-mapping necessary for

completely coalesced access. Similarly, Zhang et al. [2010] perform runtime input data rearrangement to minimize thread divergence. These two approaches can be combined with MAPS to further increase memory access efficiency in some of the memory dwarfs. The main disadvantage of these methods is that they are heavily dependent on the host CPU, which is fully utilized throughout the entire run of the GPU kernels. Furthermore, as data structures increase in size, the CPU would probably become the bottleneck of application performance.

It is worth noting that some works [Fang et al. 2014] attempt at removing shared memory optimizations entirely. This approach may be beneficial for applications that gain from L1 and L2 caching, such as the 3×3 2D image convolution shown in Section 5.4, or cache-based architectures (e.g., CPUs). Implementing this functionality as an architecture-based decision in MAPS could produce code that is not only performance portable across GPUs but also on other architectures.

7. CONCLUSION

The article demonstrated that by categorizing parallel applications to memory access patterns, it is possible to minimize memory management and indexing complexity requirements using a robust memory abstraction framework. It then presented a nontrivial CUDA implementation of MAPS and its performance on five representative applications: matrix multiplication, image convolution, all-pairs N-body simulation, cloth simulation, and fused convolution-histogram computation; each representing a different parallel programming dwarf. The results show that the performance of MAPS is comparable, and in some cases surpasses carefully optimized versions of these applications.

When manually optimizing parallel applications, it is common practice to perform profiling and static code analysis in order to find bottlenecks. During this phase, the location, amount and order of memory reads are measured in order to identify redundancy. This information can easily be used to determine which parallel container is the most suitable. Thus, programmers can speed up the process of application optimization without additional analysis of the algorithm. This remains true even for complex real-world applications, which typically contain multiple memory access patterns. Furthermore, composing parallel containers and device-level algorithms can increase the performance of fused operations by minimizing kernel overhead and omitting intermediate data output.

While extensive, the work on memory dwarfs requires further research. As the world of GPU research is constantly evolving, new memory access patterns are being discovered and used, effectively extending the current list of memory dwarfs. These new dwarfs can be implemented as containers and device-level algorithms, extending the current version of the MAPS framework.

As for input preprocessing, it can be improved by using optimizations based on works such as Dymaxion [Che et al. 2011], which reorders input data prior to launching kernels. Using the CudaDMA [Bauer et al. 2011] infrastructure in conjunction with parallel containers can also be beneficial for applications with large memory requirements.

Another area of interest is to extend our work to support multi-GPU systems and many-GPU clusters. Such systems are becoming more common, and programming them introduces more memory management challenges, such as device-to-device communication minimization, network distance optimization, and more. Extending the memory abstraction model to include more than one memory module and introducing the concept of memory distance may significantly reduce the difficulty of programming optimized multi-GPU and many-GPU applications, and benefit the developer with cleaner and more maintainable code.

ACKNOWLEDGMENT

The authors wish to thank the anonymous referees for their constructive comments.

REFERENCES

Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG'09)*. ACM, New York, NY, 145–149.

Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. 2003. STAPL: An adaptive, generic parallel C++ library. In *Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing (LCPC'01)*. Springer-Verlag, Berlin, 193–208.

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: Optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. ACM, New York, NY, Article 12, 11 pages.

Boost. 2014. Boost C++ Libraries. Retrieved from http://www.boost.org/.

Martin Burtscher and Keshav Pingali. 2011. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 75–92.

Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. 2011. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. ACM, New York, NY, Article 13, 11 pages.

CUB. 2014. CUB GPU Computing Primitives Library, NVIDIA Research. Retrieved from http://nvlabs.github.io/cub/.

CUBLAS. 2014. CUBLAS Library Documentation. Retrieved from http://docs.nvidia.com/cuda/cublas/.

CUDA. 2014. NVIDIA CUDA SDK. (2014). http://www.nvidia.com/cuda.

CUFFT. 2014. CUFFT Library Documentation. Retrieved from http://docs.nvidia.com/cuda/cufft/.

Jianbin Fang, Henk Sips, Pekka Jaaskelainen, and Ana L. Varbanescu. 2014. Grover: Looking for performance improvement by disabling local memory usage in OpenCL kernels. In *Proceedings of the 43rd International Conference on Parallel Processing (ICPP'14)*. IEEE.

Michael Garland. 2008. Sparse matrix computations on manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference (DAC'08)*. ACM, New York, NY, 2–6.

Kate Gregory and Ade Miller. 2012. *C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®*. Microsoft Press.

Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing*.

Tianyi D. Han and Tarek S. Abdelrahman. 2011. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems* 22 (2011), 78–90.

Mark Harris, John Owens, Shubho Sengupta, Yao Zhang, and Andrew Davidson. 2007. CUDPP: CUDA Data Parallel Primitives Library. Retrieved from http://gpgpu.org/developer/cudpp.

Jared Hoberock and Nathan Bell. 2010. Thrust: A Parallel Template Library. Retrieved from http://thrust.github.io/.

Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, New York, NY, 267–276.

Hugues Hoppe. 1999. Optimization of mesh locality for transparent vertex caching. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'99)*. ACM Press/Addison-Wesley, New York, NY, 269–276.

Seyong Lee and Rudolf Eigenmann. 2010. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. IEEE Computer Society, Washington, DC, 1–11.

Aaron E. Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D. Owens. 2006. Glift: Generic, efficient, random-access GPU data structures. *ACM Trans. Graph.* 25, 1 (Jan. 2006), 60–99.

MAPS. 2014. MAPS Code Repository. Retrieved from https://github.com/erdooom/MAPS.

David R. Musser, Gilmer J. Derge, and Atul Saini. 2001. *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library*. Addison-Wesley Longman, Boston, MA.

NPP. 2014. NVIDIA Performance Primitives (NPP) Library. Retrieved from http://developer.nvidia.com/npp/.

Lars Nyland, Mark Harris, and Jan Prins. 2007. Fast n-body simulation with CUDA. In *GPU Gems 3*, Hubert Nguyen (Ed.). Addison-Wesley Professional.

OpenACC. 2012. OpenACC—Directives for Accelerators. Retrieved from http://www.openacc-standard.org.

Xavier Provot. 1995. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *Graphics Interface*. 147–154.

Greg Ruetsch and Paulius Micikevicius. 2009. Optimizing Matrix Transpose in CUDA. Retrieved from https://users.csc.calpoly.edu/clupo/teaching/419/winter14/MatrixTranspose.pdf.

Pedro V. Sander, Diego Nehab, and Joshua Barczak. 2007. Fast triangle reordering for vertex locality and reduced overdraw. In *ACM SIGGRAPH. 2007 Papers (SIGGRAPH'07)*. ACM, New York, NY, Article 89.

Bjarne Stroustrup. 2000. *The C++ Programming Language* (3rd ed.). Addison-Wesley Longman, Boston, MA.

Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. 2010. Streamlining GPU applications on the fly: Thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*. ACM, New York, NY, 115–126.

Received May 2014; revised August 2014; accepted October 2014