Proceedings

2nd International Workshop on GPUs and Scientific Applications (GPUScA 2011)

Galveston Island, Texas, USA **October 10, 2011**

In conjunction with International Conference on Parallel Architectures and Compilation Techniques (PACT 2011), Galveston Island, Texas, USA, October 10-14, 2011.

Eduard Mehofer, Markus Schordan, Dan Quinlan, Beniamino Di Martino (Eds.)

Proceedings

2nd International Workshop on GPUs and Scientific Applications (GPUScA 2011)

Galveston Island, Texas, USA

October 10, 2011

Eduard Mehofer, Markus Schordan, Dan Quinlan, Beniamino Di Martino (Eds.)

Proceedings available as technical report.
Research Group Scientific Computing,
Faculty of Computer Science,
University of Vienna.
http://www.par.univie.ac.at/publications/download/TR-11-1.pdf

Technical report TR-11-1
Research Group Scientific Computing
Faculty of Computer Science
University of Vienna
October 2011

Preface

Welcome to the 2nd International Workshop on GPUs and Scientific Applications (GPUScA 2011) at Galveston Island! The workshop takes place in conjunction with PACT 2011 - the annual International Conference on Parallel Architectures and Compilation Techniques. The purpose of the workshop is to bring together GPU experts with computational science experts.

GPUs are cost-effective platforms for computational intensive applications providing tremendous peak performance. However, it is a major challenge to deliver the intrinsic performance of such architectures to end applications. The workshop addresses programming approaches and key techniques to leverage the computing power of GPUs.

Based on a double-blind peer review, 5 high-quality papers were selected for presentation and are included in the workshop proceedings. The accepted papers reflect the multidisciplinary character and the broad spectrum of the field. The presentation of the papers is arranged in two sessions.

The first technical paper session comprises two papers: 'Fast and Memory-Efficient Minimum Spanning Tree on the GPU' proposing a data-parallel implementation of Kruskal's algorithm for GPUs; 'A GPU-based Simulation for Stochastic Computing' presenting an approach which can considerably speed up the simulation time.

The second technical paper session comprises three papers: 'ForOpenCL: Transformations Exploiting Array Syntax in Fortran for Accelerator Programming' proposes a Fortran programming methodology which supports emerging computer architectures; 'Optimizing OpenCL Kernels for Iterative Statistical Algorithms on GPUs' studies the performance behavior of three important kernels; 'A Scalable Hybrid Algorithm Based on Domain Decomposition and Algebraic Multigrid for Solving Partial Differential Equations on a Cluster of CPU/GPUs' proposes a hybrid algorithm for PDEs that matches the architecture of the cluster.

It is our pleasure to announce Rudolf Eigenmann for the keynote address, whose talk is entitled 'Do GPGPUs Need Specialized Programming Environments?'. An abstract of the keynote address opens the proceedings.

After the keynote and at the beginning of the first technical paper session a round-table discussion about the topic 'Fair Evaluation of GPU Performance' shall emphasize the nature of a workshop and stimulate the scientific discourse.

The preliminary workshop proceedings are published as technical report TR-11-1 of the Research Group Scientific Computing, University of Vienna (URL http://www.par.univie.ac.at/publications/download/TR-11-1). Extended versions of the best papers will be published after the event in the International Journal of Computational Science and Engineering (IJCSE) as part of the Special Issue on: GPUs and Accelerators for Scientific Applications.

We would like to thank the program committee members and the reviewers for their hard work and the excellent cooperation. Also special thanks to all authors of submitted papers for their interest and their contributions to the success of the workshop. Finally, we are grateful to the PACT chairs for their support of the workshop.

Galveston Island, October 2011

Eduard Mehofer, Markus Schordan, Dan Quinlan, Beniamino Di Martino

Workshop Organization

Workshop Chairs

Eduard Mehofer	University of Vienna
Markus Schordan	University of Applied Sciences Technikum Wien
Dan Quinlan	Lawrence Livermore National Laboratory
Beniamino Di Martino	Second University of Naples

Program Committee

Scott Baden	University of California, USA
Uday Bondhugula	Indian Institute of Science, IND
Zhigang Deng	University of Houston, USA
Thomas Fahringer	University of Innsbruck, AUT
Dieter Kranzlmüller	Ludwig Maximilian University of Munich, GER
Beniamino Di Martino	Second University of Naples, ITA
Naoya Maruyama	Tokyo Institute of Technology, JAP
Eduard Mehofer	University of Vienna, AUT
Nacho Navarro	Barcelona Supercomputing Center, SPA
Michael O'Boyle	University of Edinburgh, GBR
Craig Rasmussen	Los Alamos National Laboratory, USA
Bernhard Scholz	University of Sydney, AUS
Markus Schordan	University of Applied Sciences Technikum Wien, AUT
Michael Schwarz	Arizona State University, USA
Xipen Shen	College of William and Mary, USA
Matthew Sottile	Galois Inc., USA
Apan Qasem	Texas State University, USA
Dan Quinlan	Lawrence Livermore National Laboratory, USA
Richard Vuduc	Georgia Tech College of Computing, USA
Michael Wimmer	Vienna University of Technology, AUT
Qing Yi	University of Texas at San Antonio, USA

Additional Reviewers

Javier Cabezas
Isaac Gelado
Klaus Kofler
Stephan Reiter

Table of Contents

Keynote

Do GPGPUs Need Specialized Programming Environments?

Rudolf Eigenmann, Professor of Electrical and Computer Engineering, Purdue University

Abstract. The talk will present the results of a recent project that explored this question by developing a translator and an automatic tuning system for OpenMP programs running on CUDA/GPGPU architectures. The talk will discuss the transformation techniques that made the most performance difference, an overview of the translator and tuning system, and the performance results of automatically translated/tuned programs versus those hand-coded in CUDA.

Fast and Memory-Efficient Minimum Spanning Tree on the GPU

Scott Rostrup, Shweta Srivastava, and Kishore Singhal
Synopsys Inc.
700 East Middlefield Rd
Mountain View, CA
rostrup@synopsys.com, shwetas@synopsys.com

ABSTRACT

The GPU is an efficient accelerator for regular data-parallel workloads, but GPU acceleration is more difficult for graph algorithms and other applications with irregular memory access patterns and large memory footprints. The Minimum Spanning Tree (MST) problem arises in a variety of applications and its solution exemplifies the difficulties of mapping irregular algorithms to the GPU. In this paper, we present a memory-efficient parallel algorithm for finding the minimum spanning tree of very large graphs by introducing a data-parallel implementation of Kruskal's algorithm. We test scalability and performance on random and real-world graphs with up to 25 million vertices and 240 million edges on an Nvidia Tesla T10 GPU with 4GB of memory. Our method can process graphs 4X larger and up to 10X faster than was possible with the recently published implementation of Boruvka's MST algorithm for the GPU. We also demonstrate the performance advantage of the proposed method against the multi-core Filter-Kruskal's MST algorithm on a dual quad-core CPU server with Nehalem X5550 processors.

1. INTRODUCTION

The graphics processing unit (GPU) is a computer architecture designed for high-throughput data-parallel applications. The massive data processing capability of the GPU is attracting researchers to explore using the GPU for general-purpose computing. Recent research has shown promising speed-ups on the GPU even for algorithms with highly irregular memory access patterns, for instance (SpMV) [4] and graph algorithms [12]. Designing efficient algorithms for the GPU requires a thorough understanding of the hardware to make most efficient use of the GPU's limited capacity, but high bandwidth graphics memory. It is especially challenging for irregular algorithms to take advantage of the GPU's high memory bandwidth since they require frequent, small memory accesses with little locality or predictability.

These observations provide a few guidelines for efficient GPU implementation of irregular algorithms, namely the algorithm should exhibit fine-grained parallelism, minimize irregular memory accesses, and keep total memory overhead low. To this end, the generic data-parallel primitives [13,19] being developed for the GPU greatly streamline the process. The efficient implementations of sort, scan, and reduction primitives let irregular graph algorithms be directly mapped onto combinations of these data-parallel primitives [23].

The MST problem on an undirected weighted graph $G = (V, E)$, with n vertices and m edges, is to find a minimally weighted subset of E that also forms a spanning tree of G. The MST problem typically arises as an approximation step to more difficult problems such as the travelling salesman problem or the minimum Steiner tree problem. There is an enormous amount of literature in the theoretical community on MST [18], including expected linear time [15] and nearly linear time algorithms [7].

Despite these results, in practice the older algorithms of Kruskal [17], Jarnik-Prim [14, 21], and Boruvka [5] still see widespread use. This is because the asymptotically more favorable approaches are more complex algorithms and hide large constants in the asymptotic analysis. Of the three older algorithms only Boruvka's algorithm is typically described in parallel. It has been implemented for a variety of parallel architectures [2,8], including the GPU in [23]. Boruvka's algorithm as implemented in [23] exploits fine-grained parallelism, but they run into memory constraints limiting them to graph sizes of 5 million vertices and 30 million edges.

An alternative approach is taken in the Filter-Kruskal algorithm [20]. It makes use of parallel edge filtering to locate edges which can be excluded from the MST quickly in parallel. Since they target commodity server architectures it is acceptable to use a sequential Union-Find data structure to merge connected components, but this will not map well to the fine-grained parallelism of the GPU.

To address these limitations we introduce the Data-Parallel Kruskal (DPK) algorithm. In order to keep memory usage low, each edge is stored once and no complex data structures are used. We partition the full problem into subproblems so that we can exploit optimal parallelism while keeping temporary storage requirements low. We implement the algorithm entirely in terms of data-parallel primitives [13] and investigate the performance of DPK on an Nvidia Tesla T10 GPU. We compare performance with both previous GPU

results and with the Filter-Kruskal's algorithm on a commodity multi-core server. Despite the algorithm's simplicity we show that on both random and real world graph instances we improve on previous work in terms of both memory usage and performance.

1.1 Related Work

Boruvka's algorithm has received a lot of attention in the literature and there are a number of different implementations for different parallel architectures. Chung and Condon [8] implement Boruvka's algorithm for distributed memory architectures and introduce a linear expected time pointer jumping algorithm. Bader and Cong [2] target shared memory processors and develop several versions of Boruvka's algorithm as well as a hybrid algorithm that mixes Jarnik-Prim's and Boruvka's algorithms. In [9], Cong, Varaswati, and Saraswat implemented a variant of Boruvka's algorithm for the IBM BlueGene architecture. There are also two existing implementations of Boruvka's algorithm for the GPU [12, 23]. Vineet et al [23] introduce an implementation of Boruvka's algorithm in terms of data-parallel primitive operations for the GPU.

Kruskal's algorithm has also attracted interest for parallel implementation for SMPs. The most straightforward approach is to perform the sorting in parallel which, in many cases, dominates runtime. Osipov, Sanders, and Singler [20] develop an alternative approach which they call Filter-Kruskal's. They make use of data-parallel primitives, sort and partition, to filter out invalid MST edges but use a sequential Union-Find data structure to merge connected components. They show good experimental speed-up on commodity servers but the sequential Union-Find makes the algorithm unsuitable for direct implementation on the GPU. Katriel, Sanders, Träff [16] provide a simplification of the randomized linear time algorithm [15] and illustrate an implementation using Jarnik-Prim on a vector machine. In contrast to these, we make use of parallel filtering but use Boruvka's algorithm instead of sequential Union-Find or Jarnik-Prim since it is best suited for massively-parallel architectures.

2. DATA-PARALLEL KRUSKAL'S (DPK)

We consider undirected weighted graphs $G = (V, E)$, with n vertices and m edges. The MST problem is to find a minimally weighted subset of E that also forms a spanning tree of G. When G is not connected, it decomposes into several MST problems, the solution to which will be a set of disjoint trees called a Minimum Spanning Forest (MSF).

Kruskal's algorithm [17] is sequential and examines one edge at a time from the lightest to the heaviest. If the edge connects two disjoint components, it is included in the MST and the two components are merged, otherwise it is discarded. Boruvka's algorithm [5] exploits the same property of considering lighter edges first but finds, in parallel, the lightest edge per vertex and connects it to a new component. Each connected component found is then compressed into a *supervertex* and the algorithm recurses, finding the lightest edges between supervertices. The problem with using Kruskal's algorithm on the GPU is that it is not a parallel algorithm. The bottom-up parallelism of Boruvka's algorithm is a good fit for the GPU and previous work has shown how to map the algorithm onto parallel architectures in terms of data-parallel primitives [2, 23].

In [23], Boruvka's algorithm is implemented using a directed adjacency list data structure which stores each undirected edge as two directed edges (see figure 1(b)). This data structure simplifies implementation but comes with a $2X$ cost in storage requirements in comparison to an undirected storage format. A second related issue is that building the directed adjacency list itself requires sorting the edges with significant associated computational cost. The directed adjacency list must be built every time the graph is contracted in each step of Boruvka's algorithm and, as observed in [2], it is the most costly part of the algorithm. Both of these problems are exacerbated by a more fundamental issue with Boruvka's algorithm. It considers all edges at every iteration, whereas it would be better to delay considering heavy edges until later.

These issues motivate us to introduce a data-parallel adaptation of Kruskal's MST algorithm that uses Boruvka's algorithm to solve subproblems in parallel on the GPU . The DPK method addresses the issue of large memory consumption by dividing the problem into subproblems with lower memory requirements. This is done by high-level partitioning of the complete graph by edge weight into subgraphs. Only these subgraphs are stored in a directed adjacency list instead of the full complete graph, resulting in both computational and memory advantage. In this section, we establish the differences among Kruskal's, Boruvka's [23] and the DPK method by giving the pseudocode for all three of them and thus highlighting the algorithmic differences resulting in memory and performance gain for the DPK method. Section 3 gives the GPU implementation details of the DPK method; performance characteristics of the algorithm are discussed in section 4.

2.1 Data Structures

To develop the algorithm we define the following data-structures:

- G: The graph stored as an undirected edge list (refer to figure 1(a)).
- $\tilde{G}$: The graph stored as a directed adjacency list (refer to figure 1(b)).
- C: Array which stores for each vertex its supervertex id.
- F: An undirected edge list that stores each MSF edge found.

2.2 Graph Primitives

Kruskal's [17], Boruvka's [5, 23], and the DPK method can be succinctly expressed in terms of a few graph primitive operations:

- **Sort**(G): Sorts the edges of G into increasing order by weight.
- **Split**(G, p): Splits G into p subgraphs G_1, G_2, ..., G_p such that the maximum edge weight in G_{k-1} is less than or equal to the minimum edge weight in G_k.

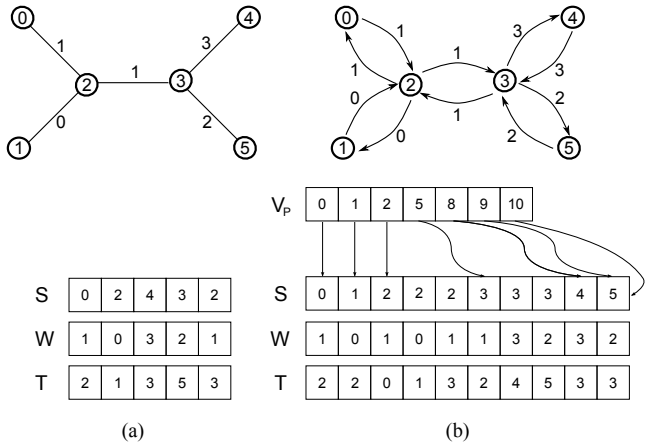


Figure 1: Two graph representations are used: (a) the undirected edge list stores each edge by source vertex id (S), weight (W), and target vertex id (T). (b) The directed adjacency list representation stores each undirected edge as two directed edges. The edges are sorted into increasing order by source vertex id and each vertex stores the index to the beginning of its list of adjacent out edges (VP).

- **BuildDirectedAdjacencyList**(G): Builds a directed adjacency list $\tilde{G}$ from an undirected edge list G.
- **ContractGraph**(G, C): Contracts G by compressing the vertices in each connected component into one supervertex. During contraction two types of edges may be filtered out:
  - Edges internal to a supervertex. In the contracted graph these are be loop edges.
  - Heavy parallel edges between supervertices. In the contracted graph this is any edge with a lighter parallel edge connecting the same two supervertices.

  Throughout contraction, each contracted edge maintains 1-1 correspondence with an edge from the original graph.
- **ConnectComponents**(F, C): For each vertex v, update $C[v]$ to the supervertex id of v's current connected component in F.
- **FindMins**(G): Finds for each vertex v in parallel, the minimum weighted edge adjacent to it. All cycles are broken by discarding edges and a set of cycle-free edges is returned.

In algorithms 1, 2, and 3 each primitive has no side-effects, as they do not alter the input arguments. The detailed implementation of these primitives on the GPU in terms of data-parallel primitives is left to section 3. The implementation may change depending on which algorithm and graph format is being used.

2.3 Algorithm Descriptions

The fundamental difference between DPK, Kruskal's and Boruvka's algorithms is the order in which edges are considered and how many edges are considered concurrently. All three of them are generic greedy MST algorithms, that is, they all consider lighter edges before heavier ones for inclusion into the MST. Kruskal's algorithm exploits no edge parallelism, it considers each edge one at a time from lightest to heaviest. At the other extreme, Boruvka's algorithm exploits maximal edge parallelism. At each iteration of Boruvka's algorithm every remaining edge is considered and the lightest edge for each supervertex is selected.

DPK resides in the middle; subgraph splitting balances between the maximal edge parallelism of Boruvka's algorithm and the sequential edge selection of Kruskal's algorithm. For instance, splitting G into m subgraphs of size 1 is Kruskal's algorithm and splitting G into 1 subgraph of size m is Boruvka's algorithm. DPK uses the subgraph splitting to control how many edges are considered concurrently and to enforce the heuristic of considering light edges before heavy ones.

```
funct Kruskal(G) → F
  F = {}
  C = [0, 1, 2, ..., n − 1]
  G = Sort(G)
  for i := 1 to m do
      (s, w, t) = G.Edges[i]
      if C[s] ≠ C[t]
        F = F ∪ {(s, w, t)}
        C = ConnectComponents(F, C)
```

Algorithm 1: Kruskal's algorithm [17].

```
funct Boruvka(G) → F
  F = {}
  C = [0, 1, 2, ..., n − 1]
  G̃ = BuildDirectedAdjacencyList(G)
  while EdgesRemaining(G̃) do
        F = F ∪ FindMins(G̃)
        C = ConnectComponents(F, C)
        G̃ = ContractGraph(G̃)
```

Algorithm 2: Boruvka's algorithm [5].

```
funct DPK(G, p) → F
  F = {}
  C = [0, 1, 2, ..., n − 1]
  {G_1, G_2, ..., G_p} = Split(G, p)
  for k := 1 to p do
      G'_k = ContractGraph(G_k, C)
      F = F ∪ Boruvka(G'_k)
      C = ConnectComponents(F, C)
```

Algorithm 3: Data-Parallel Kruskal's.

Pseudocode for Kruskal's algorithm, Boruvka's algorithm, and DPK are given in algorithms 1, 2, and 3 respectively. They are written using the same graph primitives for ease of comparison, however, this belies significant implementation, performance and memory usage differences.

2.3.1 *Kruskal's Algorithm*

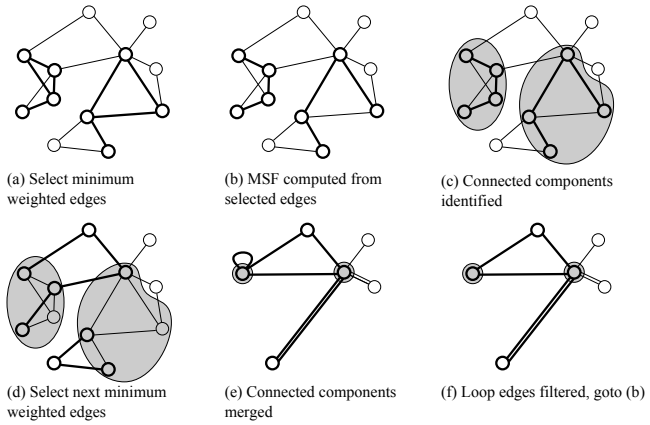


Figure 2: Schematic overview of the Data-Parallel Kruskal's algorithm. This differs from Boruvka's approach since not every edge is considered in each iteration and not all vertices will be included in each MSF.

Kruskal's algorithm can operate on undirected or directed edge lists. The initial edge list is sorted into increasing order by weight and then edges are considered from lightest to heaviest. Since each edge is considered on its own, no additional storage is needed except for storing connected component information. Maintaining and merging connected components sequentially is implemented efficiently using a Union-Find (disjoint-set) data-structure [10]. These would be used in place of the array look-ups $C[s]$ and $C[t]$ and the **ConnectComponents** primitive in our pseudocode (algorithm 1).

2.3.2 Boruvka's Algorithm

In algorithm 2 we clearly denote the transformation from undirected to directed adjacency list. We emphasize this implementation difference because its computational cost is significant and since it immediately doubles memory usage. The directed adjacency list data-structure is used because the **FindMins** graph primitive and the heavy parallel edge filtering in the **ContractGraph** step both become straightforward to implement (section 3.4) [2, 23]. The downside of this is the significant storage cost of considering all possible edges concurrently. Boruvka's algorithm requires unique edge weights for correctness, this is addressed by breaking edge weight ties with the lower target vertex id.

2.3.3 DPK

Algorithm 3 accepts an undirected edge list graph as input, the graph is then partitioned into subgraphs by edge weight using the **Split** primitive (section 3.2). Each subgraph G_k is processed sequentially from lightest to heaviest. By considering subgraphs separately the temporary storage cost is only proportional to the number of edges in the subgraph instead of the full graph. The directed adjacency list data-structure is created on the fly for each subgraph during the Boruvka's algorithm function call. The computational cost associated with the initialization is reduced as the graph is contracted before constructing the directed adjacency list. The DPK algorithm does require some additional overhead since it must maintain and merge two MSF data structures, one global MSF and one MSF internal to the Boruvka's subroutine for each subgraph.

2.4 Memory Usage

The two major factors contributing to DPK using significantly less memory than Boruvka's are the undirected edge list and the subgraph splitting. The undirected edge list stores the entire graph using 2X less memory than the directed adjacency list format. The subgraph splitting is important since several of the graph primitives use temporary storage proportional to the number of edges in the graph. Selecting the number of edges contained in a subgraph, therefore, specifies the temporary storage requirements. In section 4.2 we perform a detailed analysis of the memory usage of DPK.

2.5 Performance

The DPK approach also shows performance improvements in comparison to Boruvka's algorithm for many of the graphs tested (section 5). Graph structure and the relative computational cost of the different graph primitives are the key factors. The most critical components are the **Split** and the **ContractGraph** primitives.

The **Split** primitive is quite efficient in comparison to executing all of the graph primitives involved in a single iteration of Boruvka's algorithm. Thus by using **Split** we obtain an inexpensive heuristic able to set aside heavier edges for later consideration.

To complement this, the **ContractGraph** primitive will quickly filter out edges which become internal to an already formed supervertex. Thus heavy edges are set aside for later by **Split**, then when they are considered they are more likely to be filtered out in the **ContractGraph** primitive, than to get passed into the Boruvka's subroutine. It is significant to be able to filter out edges in the undirected edge list graph format, since **ContractGraph** is faster on an undirected edge list, than on a directed adjacency list. This is because building the contracted directed adjacency list requires sorting, but no sorting is required in the undirected case.

The role of graph structure is also important and its effect is discussed in more detail in section 4. We also investigate performance empirically across a variety of graphs in section 5.

3. DPK ALGORITHM ON GPU

We adopt the approach taken in [23] and implement our graph primitives for the GPU in terms of generic data-parallel primitives. We use the thrust data-parallel primitive library [13] and the GPU radix sort implementation of Merrill [19]. We describe how each step in algorithm 3 is mapped onto the GPU in terms of the data-parallel primitives.

3.1 Data Structures

We store the graph in an undirected edge list and denote it by $G(S, T, W)$ or simply by the symbol G in short form. It consists of three arrays S, T and W, each of length m. The arrays S, T and W store the source vertex id, target vertex

id and edge weights respectively for every edge in the graph (see figure 1(a)).

In addition to the graph array we also need to store the active subgraph G_k with m_k edges. Associated with it we keep an additional array ID of size m_k which stores for each edge in the subgraph its original position in the input graph G. The ID array maintains a one-to-one mapping between edges in the original graph and edges in contracted subgraphs.

The three global data structures G, C, and F (also defined in section 2.1) store the undirected edge list input graph, supervertex id array, and MSF respectively. We chose supervertex ids such that each supervertex is represented by a vertex number selected from one of the vertices contained in it, the *root* vertex. This simple restriction allows us to use pointer jumping [8] to update supervertex ids as new MSF edges are found.

3.2 Step 1: Split

The Split primitive acts on the input graph G to impose a partial order on the edge weights. We implement this by sorting the edges into increasing order by edge weight and then using subarrays of G as the subgraphs G_k. The sizes of the subgraphs are chosen to be $O(n)$ and increasing in size as the edge weights increase. Alternatively, and perhaps more typically, one would implement this by applying randomized partitioning recursively. However, we found for integer and floating point edge weights that the initial sorting was not a performance bottleneck (Section 5). An additional benefit of using full sorting is that once the edges are sorted, the position of the edge in the array may be used as its edge weight and its edge id. This saves memory and provides unique edge weights as required for Boruvka's algorithm to give correct results.

3.3 Step 2: Contract Graph

The subgraph G_k is contracted into G'_k, a graph between its connected components. This is implemented in two steps,

- *Relabel*: For each edge in G_k, relabel each vertex v with its supervertex id $C[v]$.
- *Filter*: Remove any edges whose source vertex id is equal to the target vertex id.

The relabeled and filtered edge list is stored in G'_k and each edge in G'_k's original position in G is stored in the array ID.

3.4 Step 3: Boruvka's Algorithm

We implement Boruvka's algorithm for the GPU using the implementation described in [23] with a few modifications. This implementation uses the directed adjacency list graph storage format (figure 1(b)). We summarize briefly the Boruvka's (algorithm 2) implementation details, but refer the reader to [23] and [2] for detailed information.

The directed adjacency list is stored in three arrays S,T, and W sorted by source vertex id. The directed edge list is then augmented with the array V_p, which stores the index into G pointing to the beginning of the local adjacency list for each vertex (see figure 1(b)). Boruvka's algorithm (algorithm 2) requires three main graph primitives,

- **FindMins**: Find the minimum weighted edge per vertex and add to the MST.
- **ConnectComponents**: Merge vertices connected in the MST thus far into supervertices.
- **ContractGraph**: Contract vertices into supervertices and rebuild the graph (Build Adjacency List).

The directed adjacency list format is well suited to the above parallel implementation since **FindMins** becomes a segmented scan. **ContractGraph** is computationally more expensive on the directed adjacency list than on the undirected edge list because of three factors. The first is that the directed format involves twice as many edges. The second is that an additional heavy parallel edge filtering phase may be applied. The third factor is the cost of building the directed adjacency list.

The most expensive part of **ContractGraph** involves sorting the edges by vertex id. Parallel edge filtering requires sorting the edges by source and target vertex id. If the parallel edge filtering is skipped then only source vertex id needs to be sorted on to build the directed adjacency list.

On output, we represent the MSF as an array of indices F_k, where each index corresponds to the edge's position in G. In addition to the edges found, we also store an array C' for each supervertex in G'_k, its new root vertex after adding the edges in F_k to F.

3.5 Step 4: Connect Components

Once new edges have been added to the MSF, the supervertex ids in C must be updated to reflect the new connections. Using the array C' computed in Boruvka's algorithm, each root vertex r updates $C[r]$ to its new root vertex. Then pointer jumping on C until the root is reached updates all vertices to their current supervertex id.

4. ANALYSIS

We consider total compute work and memory bounds in terms of n the number of vertices and m the number of edges.

4.1 Work Bound

The only routine applied to the entire graph is the **Split** primitive. Our split implementation performs $\Theta(m)$ work since it it is implemented by sorting integer or floating point edge weights using radix sort. In an edge-weight comparison model, one could instead use randomized partitioning to achieve the same result in expectation [20].

ContractGraph, **Boruvka**, and **ConnectComponents** have work complexities of $\Theta(m)$, $O(m \lg n)$, and $O(n \lg n)$ respectively. The advantage of contracting before applying Boruvka's algorithm is that we reduce the number of edges being considered by Boruvka's algorithm by taking advantage of fast parallel edge filtering in **ContractGraph**. The amount of reduction is quantified for randomly weighted graphs in section 4.3.

Boruvka's algorithm's ability to reduce the number of edges in a single iteration depends on graph structure. For instance on random graphs it doesn't reduce the number of edges very quickly [2]. This is supported by our empirical tests in section 5 for different varieties of random graphs. The worst case scenario for DPK is when filtering is largely ineffective (e.g. $m \approx n$), in that case DPK should give worse performance than Boruvka's algorithm because it does all of the same work plus the additional work of merging and maintaining two levels of MSF's.

4.2 Memory Bound

Memory usage of all three algorithms is linear in terms of the number of edges and vertices. For generality we specify the total memory usage of each in terms of implementation dependent constants α and γ which specify bytes per edge and bytes per vertex respectively. This analysis is intended to show where the memory reduction is coming from as opposed to determining an exact implementation specific bound.

The memory usages computed in equations 1, 2, and 3 do not include memory usage during the initialization phase of each algorithm. These are the **Sort**, **BuildDirectedAdjacnecyList**, and **Split** primitives for the Kruskal's, Boruvka's, and DPK algorithms respectively. This is valid under the assumption that there is enough memory to sort all of the edges.

Kruskal's algorithm has the lowest memory usage of the three,

$$\mathrm{M}_K(m,n) = \alpha_K m + \gamma_K n. \quad (1)$$

It requires only the storage of the undirected edge list plus the space for the Union-Find data-structure.

The memory usage of Boruvka's algorithm [23] using the directed adjacency list (i.e $2m$ edges) may be written as,

$$\mathrm{M}_B(m,n) = 2\alpha_B m + \gamma_B n. \quad (2)$$

The α_B constant includes storage of each edge plus temporary space used in the various graph primitives in Boruvka's algorithm [23].

The DPK approach limits the edge input size to the Boruvka's bound (equation 2) by applying it only to subgraphs. For simplicity we consider an even splitting of the graph into p subgraphs, each with m/p edges. In that situation the memory usage of DPK can be written,

$$\mathrm{M}_{DPK}(m,n,p) = \alpha_D m + \gamma_D n + \mathrm{M}_B(\frac{m}{p}, n). \quad (3)$$

DPK requires only the storage of the undirected edge list since no primitives act on all edges after the **Split** primitive. The edge component of equation 2 is then reduced by the factor p. The DPK approach does duplicate some storage in the γ factors. However, only for very sparse graphs ($m \approx n$), would we expect the γ factors to be dominant.

Comparing α factors, we see that α_K and α_D are simply the space used to store a single edge, while α_B must account for any temporary storage used within the graph primitives. This can be very significant when using the data-parallel primitive approach, but is dependent on the implementations used. Rather than examine each primitive implementation analytically, we use the same library [13] to implement both and compare usage empirically in section 5.

4.3 Analysis for Random Graphs

To improve on the Boruvka's algorithm upper bound of $O(m \lg n)$, we adopt the approach taken in [20] and restrict the graphs we consider to those with random edge weights. The effect of considering only graphs with randomly assigned edge weights is that we can now consider the algorithm as a variant of the linear expected time algorithm [15] since a random sample of r edges from a graph is equivalent to taking the lightest r edges from a graph with randomly assigned edge weights. This will give us expected bounds on the number of edges remaining after filtering.

Applying the sampling lemma from [6,15] to randomly weighted graphs gives that, if we select the lightest r edges of a graph with random edge weights, the probability that another edge does not form a cycle when added to the MSF of those r edges is n/r. This is equivalent to saying that the probability that an edge will not be filtered out in the **ContractGraph** step is equal to n/r.

In [20] they use this idea to count the number of comparisons used in the quicksort algorithm to obtain an expected bound on the work done. We can apply a simpler analysis to bound the amount of work done in the Boruvka's algorithm portion of DPK on a randomly weighted graph. We assume our edge list has been sorted and select p subgraphs with increasing sizes of $n, n, 2n, 4n, ..., m/2$, where for simplicity, we assume dummy edges are added to pad the graph out until $m = 2^p n$, thus $p = \lceil \lg(m/n) \rceil$. By applying the sampling lemma we know that given a sample of size $m/2$, if we apply the filter to the next $m/2$ edges, then we expect n edges to survive the filtering. We apply no filtering to the first subgraph, and then apply the sampling lemma thereafter, thus each subgraph is expected to contain n edges. If we bound the maximum number of Boruvka iterations by $\lg n$ then we can place an upper bound on the expected total work done inside the p Boruvka's iterations as,

$$W_B(n,m) = O(pn \lg n). \quad (4)$$

Combining the Boruvka's expected work bound with the $O(m)$ subgraph splitting bound and replacing p with $\lg(m/n)$ gives an upper bound on the expected work,

$$W(n,m) = O(m + n \lg n \lg \frac{m}{n}). \quad (5)$$

We thus expect to see linear performance in the number edges provided that the graph is sufficiently dense. One missing link from this analysis is that inside each Boruvka iteration, pointer jumping is used to merge vertices into supervertices and it has worst case runtime of $O(n \lg n)$. There is a known expected linear time randomized pointer jumping algorithm [8]. However, we used straightforward pointer jumping and found it to only be a small contributor to the total runtime (see Boruvka's algorithm in table 1).

5. EXPERIMENTS

We compare our implementation of DPK against the Boruvka's algorithm implementation from [23] on the GPU. We

also compare the DPK performance with that of a sequential implementation of Kruskal's algorithm and a competitive multi-core MST algorithm, Filter-Kruskal's [20] on an x86 server.

5.1 Setup

5.1.1 GPU

We investigate the performance of the DPK method on a Tesla T10 GPU with 4GB of memory, from one quarter of an S1070 GPU computing rack and compare it against Boruvka's algorithm [23] on the same Tesla T10 GPU. We implement both algorithms using the data-parallel primitives of the thrust (v 1.5) library [13]. We compile our CUDA code using cuda 3.2 on optimization level -O3 targeting Nvidia devices of compute capability 1.3.

5.1.2 Multi-core CPU

We implement the sequential Kruskal's method [17] and the multi-core Filter-Kruskal's method [20] for a dual-socket quad-core Intel Nehalem X5550 system (@2.67GHz). The multi-core Filter-Kruskal method [20] is similar in design to our algorithm 3. They use randomized recursive partitioning instead of sorting to do the subgraph splitting and use Kruskal's algorithm to find MSF's. Kruskal's algorithm uses a sequential Union-Find data structure for merging connected components. We implement the Filter-Kruskal algorithm using the same gnu libstdc++ parallel mode [22] data-parallel primitive implementations as used in [20] and use the boost (v 1.45) disjoint sets implementation for the Union-Find data structure. We compile with gcc (v 4.3.4) using the optimization flags -O3 -march=native on the native host machine.

5.1.3 Algorithm Parameters

The Boruvka's [23], DPK and Filter-Kruskal's [20] methods contain a few parameters:

- $b = 4$: Number of Boruvka steps in [23] before parallel edge filtering is applied.
- $z = 2$: Scaling Factor in the DPK method. The k^{th} subgraph size is taken to be the minimum of $z^{k-1}n$ and the number of remaining edges (section 4.3).
- Recursive subproblems in Filter-Kruskal's [20] are at most of size $2n$.

For the input graphs we use 32-bit floating point edge weights and 32-bit unsigned integer vertex ids. The only exceptions are two optimized implementations that trade edge weight accuracy for storage. The DPK 10-bit and Boruvka's 10-bit [23] implementations map edge weights to 10-bit integers and pack source vertex, target vertex and edge weight into one 64-bit word. One other comment is that we include the time spent building an initial directed adjacency list in the total runtimes reported for Boruvka's algorithm [23] on the GPU.

5.2 Random Graphs

We make use of three different sets of random graph distributions from the Georgia Tech Graph Generator suite [3] to test the performance and scalability of the different algorithms. The three distributions are the Erdosz-Renyi random graph model (RAND), the RMAT power-law degree distribution model, and the SSCA2 hierarchical clique model. We create two types of random graphs scaling up to 240 million edges:

1. *Fixed Vertex Degree (FVD)*: Keep the average vertex degree fixed while scaling the number of vertices and edges,
2. *Fixed Vertex Number (FVN)*: Keep the number of vertices fixed and scale the graph size by adding more edges, thus increasing the average vertex degree.

5.2.1 Fixed Vertex Degree

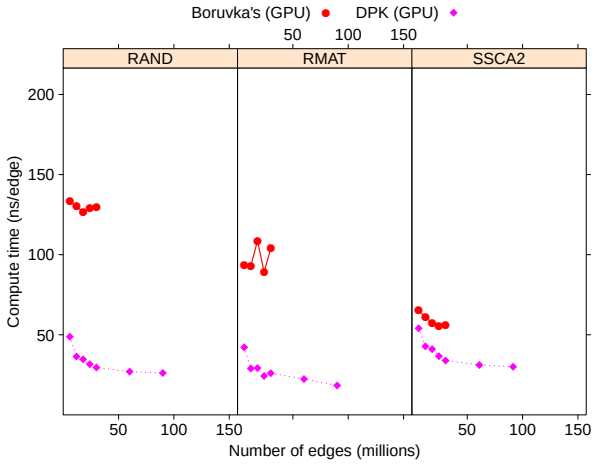
In figure 3, we show the results for FVD graphs with average vertex degree fixed at 6. The DPK method is compared against Boruvka's [23] in figure 3(a) and DPK 10-bit is compared with Boruvka's 10-bit [23] in figure 3(b). From figure 3(a) and figure 3(b), it is clear that the DPK method performs 1.2X-5X faster than the Boruvka's [23] while also being able to solve problems 2.5X-3X larger for the random FVD graphs.

In figure 4, performance on the FVD graphs is shown for the sequential CPU implementation of Kruskal's [17] and the 8-core Filter Kruskal's [20]. They are compared against the DPK 10-bit GPU implementation. Both CPU implementations seem to scale with the number of vertices. In the plot, we see that on the smaller graphs the CPU implementation is faster, however on the larger problem sizes we see the GPU implementations' time per edge flatten out and perform better than the 8-core CPU implementation. This arises from the fact that the Union-Find data structure used to keep track of connected components uses frequent scattered memory accesses and is not cache friendly. Thus as n increases the performance of this data structure degrades. On the GPU, however, scattered memory accesses always incur a fixed performance penalty regardless of the number of vertices.

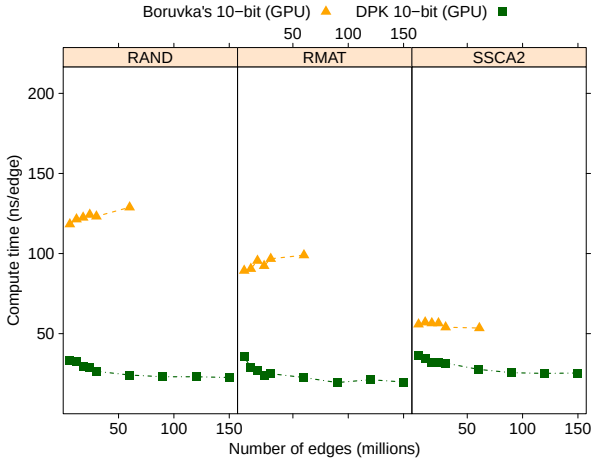
5.2.2 Fixed Vertex Number

In figure 5, we show the results for FVN graphs of increasing number of edges and the number of vertices fixed at one million. The DPK method is compared against Boruvka's [23] in figure 5(a) and the DPK 10-bit is compared against Boruvka's 10-bit [23] in figure 5(b). From figure 5(a) and figure 5(b), we clearly see that the DPK method outperforms the Boruvka's [23] for all sizes of random FVN graphs both in terms of timing and memory efficiency. The DPK method is 1.2X-10X faster and processes 4X larger graphs than is possible with Boruvka's [23]. As the number of edges increases, performance flattens out, showing that the algorithms performance is scaling linearly in the number of edges.

In figure 6, performance of the FVN graphs is shown for the sequential CPU implementation of Kruskal's [17] and 8-core Filter-Kruskal's [20]. They are compared with DPK 10-bit on the GPU. In this case, with the number of vertices held constant, we see that both Filter-Kruskal's [20] and DPK scale linearly in the number of edges. Interestingly, the performance between the DPK and the Filter-Kruskal's [20] becomes almost identical as graph density increases.



(a) DPK vs Boruvka's.



(b) DPK vs Boruvka's with packing optimization.

Figure 3: (FVD) Performance comparison on random graphs with average vertex degree 6 and increasingly many vertices and edges. Performance is compared between DPK and Boruvka's [23] on a Tesla T10 GPU with 4GB of memory. (a) DPK and Boruvka's use 32-bit floating point edge weights. (b) DPK and Boruvka's use packed 10-bit integer edge weights. The number of vertices and edges begins at one million and six million and is increased until memory capacity is exceeded on the GPU.

5.2.3 *Detailed Runtimes*

In table 1 we examine where each algorithm spends its runtime. Examining Boruvka's algorithm performance we see that it is not competitive with the other two algorithms and that the dominant component is the **ContractGraph** primitive. This is because **ContractGraph** requires rebuilding a directed adjacency list and on random graphs Boruvka's [23] does not decrease the number of edges per iteration very quickly [2]. We also see that the **BuildDirectedAdjacencyList** initialization cost alone forms roughly a sixth of the runtime.

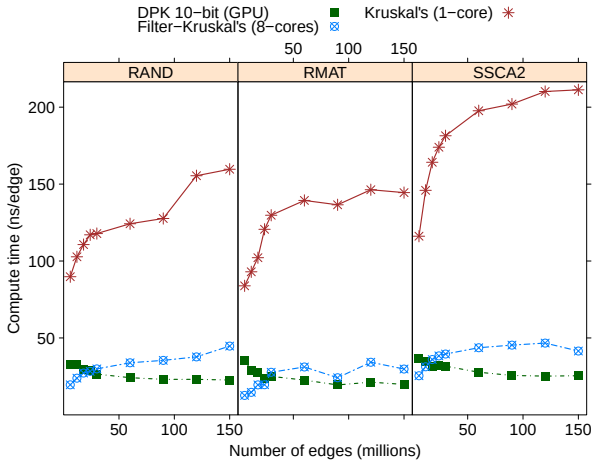


Figure 4: (FVD) Performance comparison on random graphs with average vertex degree 6 and increasingly many vertices and edges. Performance is compared between the DPK method with packing optimization on a Tesla T10 GPU with 4GB of memory, a sequential CPU implementation of Kruskal's algorithm and Filter-Kruskal's [20] on 8-cores. The number of vertices and edges begins at one million and six million and is increased until memory capacity is exceeded on the GPU.

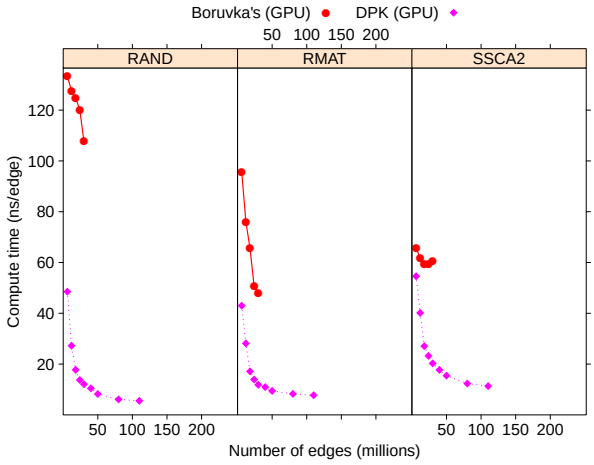
Both DPK and Filter-Kruskal's take advantage of the fast **ContractGraph** primitive on undirected edge lists to filter out loop edges. However, the sequential Union-Find component of Filter-Kruskal's is a limitation to scalability, as it already comprises 58% and 38% of the runtime in tables 1(a) and 1(b) respectively.

5.3 Real World Graphs

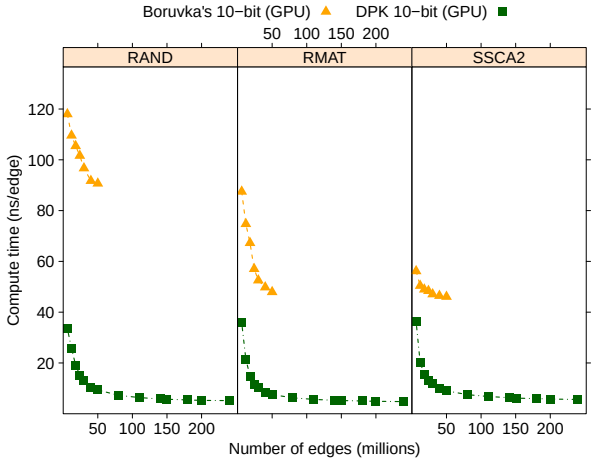
We selected sparse matrices from the Florida Sparse Matrix Collection [11] that represent networks with different graph properties and that arise from different application domains (see table 2). We mapped directed graphs onto undirected graphs and if edge weights were not provided we used uniformly random floating point edge weights. In figure 7 we arrange the graphs from top to bottom by increasing average vertex degree. We compare the two 10-bit edge weight optimized GPU implementations with the Filter-Kruskal's [20] 8-core implementation. All three algorithms seem to show better performance as the average vertex degree increases with DPK showing the best GPU performance and Filter-Kruskal's the best performance on the majority of the graphs.

Comparing the two GPU implementations, only on the extremely sparse graphs (table 2) do we see Boruvka's outperforming DPK. This is because for very sparse graphs DPK is essentially Boruvka's algorithm with some additional overhead. The sparsest graphs are also some of the smaller networks investigated with less than 10 million edges each. On all other graphs we see DPK outperforming Boruvka's algorithm.

6. CONCLUSION



(a) DPK vs Boruvka's.



(b) DPK vs Boruvka's with packing optimization.

Figure 5: (FVN) Performance comparison on random graphs with a fixed number of vertices and increasingly many edges. Performance is compared between DPK and Boruvka's [23] on a Tesla T10 GPU with 4GB of memory. (a) DPK and Boruvka's use 32-bit floating point edge weights. (b) DPK and Boruvka's use packed 10-bit integer edge weights. The number of vertices is fixed at one million and the number of edges varies from one to 240 million.

In this paper we introduced the Data-Parallel Kruskal's MST algorithm. We illustrated how it can be implemented for current GPU architectures using data-parallel primitives and investigated its theoretic and memory advantages over using Boruvka's algorithm. We also demonstrated these results experimentally by finding the MST of a variety of large graph problems using 4X less memory and up to 10X faster than possible with Boruvka's algorithm on the GPU. We also compared performance against a modern parallel MST algorithm for multi-core commodity server architectures and showed competitive performance ratios. As increasingly many cores are placed onto a single die, it will become difficult for algorithms with sequential bottlenecks to keep up. The constrained parallelism of our approach gives

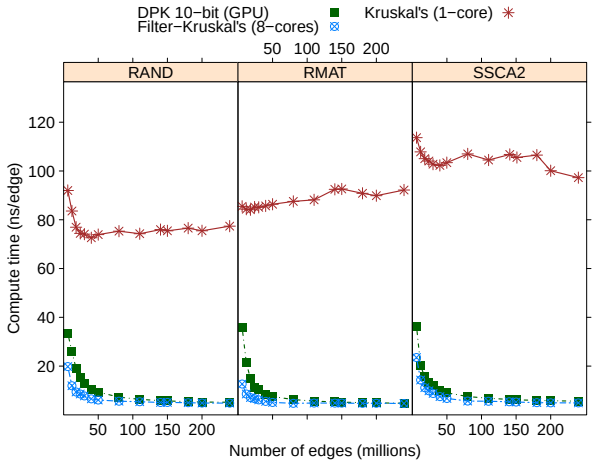


Figure 6: (FVN) Performance comparison on random graphs with a fixed number of vertices and increasingly many edges. Performance is compared between DPK with 10-bit edge weight packing optimization on a Tesla T10 GPU with 4GB of memory, a sequential CPU implementation of Kruskal's algorithm and Filter-Kruskal's [20] on 8-cores. The number of vertices is fixed at one million and the number of edges varies from one to 240 million.

(a) Runtime in milliseconds on random graphs with 5 million vertices and 30 million edges.

Component	Boru	DPK	FK8
Split/Partition	-	99	116
FindMins/Sort	554	-	46
ContractGraph	2745	124	211
BuildDirectedAdjacencyList	482	171	-
ConnectComponents	120	83	-
Boruvka's/Union-Find	-	434	524
Total	3901	911	897

(b) Runtime in milliseconds on random graphs with 1 million vertices and 30 million edges.

Component	Boru	DPK	FK8
Split/Partition	-	99	84
FindMins/Sort	554	-	8
ContractGraph	2204	26	53
BuildDirectedAdjacencyList	448	39	-
ConnectComponents	30	15	-
Boruvka's/Union-Find	-	169	87
Total	3236	348	232

Table 1: Component-wise performance comparison on two random graphs with the same number of edges, but different number of vertices. Performance is compared among Boruvka's [23] (Boru) on a Tesla T10 GPU, DPK on a Tesla T10 GPU, and Filter-Kruskal's [20] on 8-cores (FK8). The component graph primitives are as listed in the pseudocodes in algorithms 2 and 3. Each of the split row titles: Split/Partition, FindMins/Sort, Boruvka's/Union-Find specify the first and second entries in their row respectively.

Name	n (M)	m (M)	Avg. Deg.
roadNet-TX	1.39	1.9	1.4
roadNet-CA	1.97	2.8	1.4
roadNet-PA	1.09	1.5	1.4
Freescale1	3.43	8.5	2.5
atmosmodl	1.49	4.4	3.0
kkt_power	2.06	6.5	3.1
wb-edu	9.85	46	4.7
circuit5M	5.56	27	4.9
thermomech_dK	0.20	1.3	6.5
socLiveJournal1	4.85	43	8.8
bone010_M	0.99	11	12
wikipedia-20070206	3.57	42	12
nlpkkt120	3.54	47	13
af_shell10	1.51	25	17
bone010	0.99	35	36
audikw_1	0.94	38	41
RM07R	0.38	20	52
mouse_gene	0.05	14	320

Table 2: Graphs selected from the Florida Sparse Matrix Collection [11]. Also listed is the number of vertices n (in millions) and the number of undirected edges m (in millions), and their average vertex degree. The results of computing the MST on these graphs is shown in figure 7.

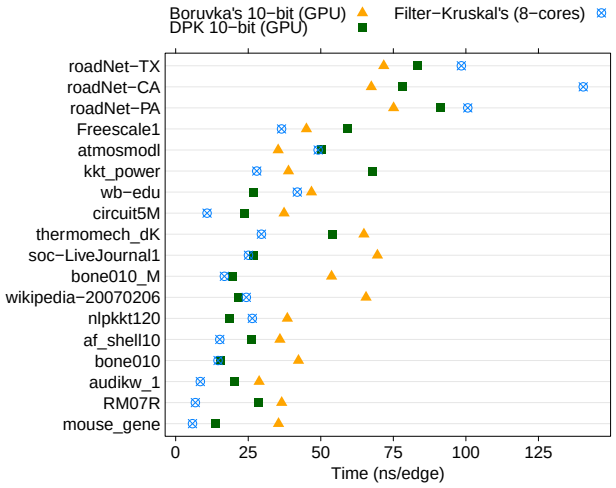


Figure 7: Runtime per edge is plotted for a variety of graphs from the Florida Sparse Matrix Collection [11] (see table 2). The performance of the Data-Parallel Kruskal's algorithm with packed edge weights is compared to the packed edge weight Boruvka's algorithm on the GPU [23] and to Filter-Kruskal's [20] on an 8-core server machine. The graphs are arranged from sparsest at the top to the most dense at the bottom.

a practical approach to implementing an efficient MST algorithm for GPU. An interesting avenue for further research would be to add randomization to the algorithm and to use a GPU implementation of ranged-minimum-queries, similar to [16]. This would eliminate the upfront cost of sorting the edges by weight and eliminate any weaknesses to particular worst-case graphs. To accommodate yet larger graphs DPK could also be modified to stream data to one or multiple GPUs, similar to the external memory MST formulation in [1].

7. REFERENCES

[1] J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external memory graph algorithms. In Proceedings of the 6th ESA, Venice, Italy, 1998.

[2] D. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. Journal of Distibuted Computing, 66:1366–1378, 2006.

[3] D. A. Bader and K. Madduri. Gtgraph: A synthetic graph generator suite. Technical Report GA 30332, Georgia Institute of Technology, 2006.

[4] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pages 1–11, New York, NY, USA, 2009. ACM.

[5] O. Borùvka. O jistém problému minimálním. Práce mor. pøírodovìd. spol., 3:37–58, 1926.

[6] T. M. Chan. Backwards analysis of the Karger-Klein-Tarjan algorithm for minimum spanning trees. Information Processing Letters, 67(6):303–304, 1998.

[7] B. Chazelle. A minimum spanning tree algorithm with Inverse-Ackermann type complexity. Journal of the ACM, 47:1028–1047, 2000.

[8] S. Chung and A. Condon. Parallel implementation of Boruvka's minimum spanning tree algorithm. In IPPS, 1996.

[9] G. Cong, G. Almasi, and V. Saraswat. Fast pgas implementation of distributed graph algorithms. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. MIT Press, third edition, 2009.

[11] T. A. Davis and Y. Hu. University of Florida Sparse Matrix Collection. Technical report, University of Florida, 1997.

[12] P. Harish, V. Vineet, and P. J. Narayanan. Large graph algorithms for massively multithreaded architectures. Technical Report IIIT/TR/2009/74, International Institute of Information Technology Hyderabad, 2009.

[13] J. Hoberock and N. Bell. Thrust: A parallel template library, 2010. Version 1.4.

[14] V. Jarník. O jistém problému minimálním. Práce Moravské Pøírodovìdecké Spoleènosti, 6:57–63, 1930.

[15] D. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. Journal of the ACM, 42:321–329, 1995.

[16] I. Katriel, P. Sanders, and J. Traeff. A practical minimum spanning tree algorithm using the cycle

property. In 11th European Symposium on Algorithms (ESA), volume 2832, pages 679–690. LNCS, 2003.

[17] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. In Proceedings of the American Mathematical Society, pages 48–50, 1956.

[18] M. Mares. The saga of minimum spanning trees. Computer Science Review, 2(3):165 – 221, 2008.

[19] D. Merrill and A. Grimshaw. Revisiting sorting for GPGPU stream architectures. Technical Report CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA, 2010.

[20] V. Osipov, P. Sanders, and J. Singler. The Filter-Kruskal minimum spanning tree algorithm. In ALENEX, pages 52–61, 2009.

[21] R. C. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, 36:1389–1401, 1957.

[22] J. Singler, P. Sanders, and F. Putze. Mcstl: The multi-core standard template library. In A.-M. Kermarrec, L. Bouge, and T. Priol, editors, Euro-Par 2007 Parallel Processing, volume 4641 of Lecture Notes in Computer Science, pages 682–694. Springer Berlin / Heidelberg, 2007.

[23] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In HPG '09: Proceedings of the Conference on High Performance Graphics 2009, pages 167–171, New York, NY, USA, 2009.

A GPU-based Simulation for Stochastic Computing

Peng Li, Weijun Xiao, and David J. Lilja
Department of Electrical and Computer Engineering
University of Minnesota, Twin Cities
Minneapolis, MN, 55455
{lipeng, wxiao, lilja}@umn.edu

ABSTRACT

Stochastic computing performs operations using streams of bits that represent probability values instead of deterministic values. An important benefit of stochastic computing is that it can tolerate a large number of failures in a noisy system. Additionally, for the VLSI implementation of a sophisticated algorithm, a stochastic implementation can consume much less hardware with lower power compared to a deterministic implementation with comparable performance. However, the simulation of the stochastic implementation is extremely time consuming when it is run on a conventional processor. This is because a probabilistic value is represented by a long bit stream (for example, 8192 bits) in stochastic computing. The simulation time of the stochastic implementation is normally hundreds or thousands of times slower than the deterministic implementation for the same algorithm. In this paper, we propose a GPU-based solution which can greatly speed up the simulation of stochastic computing. To validate our GPU-based simulation, we use the stochastic implementation of the frame difference-based image segmentation algorithm as an example to conduct extensive experiments. Measured results show that our GPU-based simulation can achieve up to 119 times performance speedup compared to the CPU-based simulation.

1. INTRODUCTION

Stochastic computing, which has been known for a long time [6], performs operations using probabilistic values instead of deterministic values. The initial motivation for researchers working on this technique was to take advantage of its low hardware cost. For example, a multiplication needs only an AND gate in stochastic computing, while an addition and a subtraction can be performed using a multiplexer, and a division can be implemented using a J-K Flip-Flop. Additionally, Brown and Card [3, 4] found that a Finite State Machine (FSM) can be used to implement some sophisticated functions (such as exponential and tanh functions) in stochastic computing using less hardware than combinational logic. For example, their stochastic exponential function needs only four D Flip-Flops. Furthermore, Qian et al [15, 14] proposed a novel synthesis approach that utilizes Bernstein basis functions to approximate polynomial functions in stochastic computing. In addition to the low hardware cost, another advantage of stochastic computing is its high level of fault-tolerance. Because stochastic computing performs operations using probabilities instead of deterministic values, it can gracefully tolerate a very large numbers of errors compared to the Triple Modular Redundancy (TMR) techniques while maintaining the equivalent performance. Previous research has shown that stochastic computing can tolerate much higher noise and soft errors for digital image processing applications than conventional implementations [14, 9]. For instance, Qian et al [14] showed that the stochastic implementation of an image gamma correction algorithm can gracefully tolerate a very large number of errors compared to a deterministic implementation. In our previous work [9], we showed that the stochastic implementation of a KDE-based image segmentation algorithm [5] can still have very good segmentation quality compared to a TMR implementation when the soft error rate reaches up to 30%.

To take advantages of these benefits, a stochastic computing system must be implemented in hardware, such as an FPGA or ASIC. A behavior level simulation performed on a general purpose processor is a necessary procedure before the hardware implementation. However, the simulation time of a stochastic implementation is extremely long even for a simple algorithm. In Table 1 we show the simulation times of the deterministic implementations and the stochastic implementations of the frame difference-based image segmentation algorithm based on five different image resolutions. Table 2 lists the detailed specification for the CPU we used in our experiments. It can be seen that the simulation time of the stochastic implementations is much longer. This is mainly because stochastic computing uses a long bit stream (8192 bits in this example) to represent a probability value. Thus, the simulation time of the stochastic implementation is normally hundreds or thousands of times slower than the deterministic implementation for the same algorithm. Fortunately, the rapid development of Graphic Processing Units (GPUs) brings a new opportunity to solve the computation bottleneck of simulating stochastic computing elements. Due to the low cost and parallel computing capability of the GPUs, GPU computing has recently been used for a wide range of high-performance computing applications [16, 13, 11, 8]. Benefiting from the GPU hardware, applications can often achieve more than 100 times performance speedup

Table 1: Simulation times of the two implementations of the frame difference-based image segmentation algorithm for five different image resolutions.

Resolution	Run time of the deterministic implementation	Run time of the stochastic implementation
176 × 144	0.0013s	9.45s
352 × 288	0.0055s	37.14s
640 × 480	0.0237s	112.55s
1280 × 720	0.0755s	337.82s
1920 × 1080	0.1701s	753.83s

Table 2: Main features of the CPU used for the experiments.

CPU Model	Intel Duo E8400
Number of multiprocessors (SM)	1
Number of cores	2
SM clock frequency	3.0 GHz
Memory Frequency	1.33 GHz
L1 Cache	2×32KB/2×32KB
L2 Cache	6MB

for digital signal processing, physical simulations, biomedical imaging, geologic computation [12, 17, 2], and other fields. In this paper, we investigate using the GPUs to reduce the simulation time of stochastic computing. Using the frame difference-based image segmentation algorithm as a case study, we have conducted extensive experiments to check the potential computation capability of the GPUs for simulating stochastic computing. Our experimental results indicate that the GPU-based simulation approach can obtain the results over 119 times faster than the CPU setting, which shows great promise for using the GPUs to reduce the simulation time of stochastic computing. To the best of our knowledge, this is the first work to apply GPU technology to stochastic computing.

The remainder of this paper is organized as follows. Section 2 briefly introduces the background of stochastic computing and the GPU hardware and software. We present the basic Stochastic Computational Elements (SCEs) and the corresponding CUDA implementations for the GPU-based simulation in Section 3. Section 4 uses the frame difference-based image segmentation algorithm as a case study to show how to use the GPU to speed up the simulation time of a stochastic computing system. Section 5 describes the experimental methodology and measurement results with conclusions drawn in Section 6.

2. STOCHASTIC COMPUTING AND THE GPU HARDWARE AND SOFTWARE

2.1 Stochastic Computing

Stochastic computing performs operations using probabilities instead of deterministic values, i.e., computations in the deterministic boolean domain are transformed into probabilistic computations in stochastic computing [6, 3]. In this section, we briefly describe the background of stochastic computing including the coding formats and conversion approach between a deterministic value and a stochastic bit stream.

2.1.1 Coding Formats

A stochastic computing system can have two possible coding formats, a unipolar coding format and a bipolar coding format. In the unipolar coding format, a real number x in the unit interval (i.e., $0 \leq x \leq 1$) corresponds to a bit stream $X(t)$ of length L, where $t = 1, 2, ..., L$. The probability that each bit in the stream is one is

$$P(X = 1) = x. \quad (1)$$

For example, the value $x = 0.3$ would be represented by a random stream of bits such as, 0100010100, where approximately 30% of the bits are "1" and the remainder are "0".

In the bipolar coding format, the range of a real number x is extended to $-1 \leq x \leq 1$, however, the probability that each bit in the stream is one is

$$P(X = 1) = \frac{x + 1}{2}. \quad (2)$$

These two coding formats are the same in essence, and can coexist in a single system. The trade-off between these two coding formats is that the bipolar format can deal with negative numbers directly while, given the same bit stream length, L, the precision of the unipolar format is twice that of the bipolar format.

2.1.2 Conversion Approach

To convert a deterministic value x_d ($x_d \in [a, b]$) into a stochastic bit stream X, we can generate a random number and compare it to x_d. The pseudocode of this operation is shown as follows,

```
for (i = 0; i < L; i++){
    if rand() < (xd - a) / (b - a)
        X(i)=1;
    else
        X(i)=0;}
```

where L is the length of the stochastic bit stream X. The function $rand()$ is used to generate a random number in the range $[0, 1]$ based on a uniform distribution. The stochastic bit stream X generated by this code has the probability

$$P(X = 1) = \frac{x_d - a}{b - a}. \quad (3)$$

By counting the number of "1" bits in a stochastic bit stream, we can convert the stochastic bit stream back to the corresponding deterministic value as follows,

$$x_d = a + \frac{sum(X) \cdot (b - a)}{L}. \quad (4)$$

2.2 The GPU Hardware and Software

In this paper, we leverage the high-performance capability of the NVIDIA GPUs to speed up the simulation of a stochastic

computing system. As we know, the performance improvement for a single-threaded processor has almost come to an end because we are hitting the memory and energy walls and facing great challenges for increasing the clock frequency and the transistor density. The performance of applications can only be improved if parallelism can be utilized for them. Today's GPUs with hundreds of parallel processor cores deliver parallel performance much more efficiently than CPUs and achieve significant performance speedup by executing tens of thousands of parallel threads. For example, the NVIDIA Geforce 8800GTX can achieve up to 518 Gflops peak performance by running 12,288 total threads spread across 128 cores.

Table 3: Main Features of NVIDIA Tesla C2050 used for the experiments

GPU Model	Tesla C2050
Number of multiprocessors (SM)	14
Number of cores	448
SM clock frequency	1.15 GHz
Single Precision peak GFLOPS	1030
Double Precision peak GFLOPS	515
Max number of thread per block	1024
Number of registers per SM	32768
Memory Frequency	1.5 GHz
Memory Bandwidth	144 GB/s
Memory ECC	Yes
Memory Interface	384 bit
Shared Memory	16KB/48KB
L1 Cache	48KB/16KB
L2 Cache	768KB

The recently introduced Fermi, which is NVIDIA's latest GPU architecture, can provide much better performance and memory bandwidth than previous-generation GPUs. The Fermi architecture enables us to achieve higher performance speedup and to solve larger problems than we have been able to solve previously. It includes several new developments over the previous GPUs including more computing units and a new memory hierarchy. The NVIDIA Fermi GPU is composed of an array of streaming multiprocessors (SM), each of which has 32 scalar processors housed in two scalar processor clusters. Because each cluster works on an individual warp and issues a new instruction every two cycles, a SM needs at least 2x32 threads to be fully utilized and maximally allows 1,536 threads to be active simultaneously. All these threads in a kernel execute the same instructions on different sets of data. A block of threads is mapped to and executed on a SM. The threads within the block are grouped by multiple warps for scheduling. In each warp, 32 threads will be executed in a lock step. For the memory hierarchy, the Fermi GPUs introduced L1 and L2 caches. For each SM, the 64KB on-chip memory can be split into a 48KB shared memory and a 16KB L1 cache or into a 16KB shared memory and a 48KB L1 cache. Depending on different data access patterns, users can dynamically choose using either a big or a small L1 cache for performance optimization. The L1 cache is distributed and associated with a SM while the L2 is a unified cache to reduce the long latency of accessing global memory. The size of the L2 cache for the current Fermi GPUs is 768KB. Table 3 lists the detailed specification of the Fermi-based Tesla C2050 GPU we used in our experiments.

For the GPU programming, we use CUDA C to conduct our experiments. CUDA embeds the GPU code inside C code, using the language extensions to indicate whether a function should be executed on the CPU (called *host*) or on the GPU (called *device*). CUDA hides all architectural details (such as threads, warps, SM, etc.) to users and instead exposes the logical notions of blocks, grids, and threads to help the decomposition of the problem domain.

3. CUDA IMPLEMENTATIONS OF THE BASIC SCES

In order to simulate a stochastic computing system with the GPU-based parallel computing, we need to implement the basic stochastic computing operations with CUDA first. In this section, we will introduce the basic SCEs in stochastic computing, including both the combinational logic-based SCEs and the FSM-based SCEs. Additionally, we demonstrate the CUDA implementation of each of them. Our objective is to use these CUDA-based functions to develop a general simulator for a stochastic computing system.

3.1 Initialization

In a stochastic computing system, we use a random bit stream to represent a deterministic value. These random bit streams must be independent of each other to ensure no correlation between stochastic bit streams. In our CUDA implementation, we use the CURAND library to generate random bit streams. By assigning the same seed and the unique sequence number for each of these streams, we can guarantee that they are independent of each other [1]. The following CUDA code shows the kernel function, *setup_kernel*, for initialization.

```
1 __global__ void setup_kernel(curandState *state) {
    int idx;
3
    idx = threadIdx.x + blockIdx.x * blockDim.x;
5   if (idx < MAXSTREAMS)
      curand_int(1234, idx, 0, &state[idx]); }
```

In this code, each thread handles a single random bit stream. A global array *state[]* is allocated to record the random state for each thread. **MAXSTREAMS**, which is the total number of streams, depends on the number of input data of a system. For example, we need to generate $640 \times 480 = 307{,}200$ random bit streams for the stochastic implementation of the image gamma correction algorithm if the resolution of the input images is 640×480.

3.2 The Combinational Logic-Based SCEs

Three basic operations - scaled addition, scaled subtraction, and multiplication - can be implemented using combinational logic in stochastic computing [6, 3].

3.2.1 Scaled addition

Scaled addition can be implemented with a multiplexer (MUX) for both bipolar and unipolar coding format. The function of the MUX is,

$$C = S \cdot A + (1 - S) \cdot B. \quad (5)$$

If we assume A, B, C, and S are stochastic bit streams, and P_A, P_B, P_C, and P_S stand for their corresponding probabilities, we will get

$$P_C = P_S \cdot P_A + (1 - P_S) \cdot P_B. \quad (6)$$

In (6), we normally set $P_S = 0.5$ to perform unbiased scaled addition.

3.2.2 Scaled subtraction

Scaled subtraction can be implemented with a MUX and a NOT gate It works only for the bipolar coding format. The function of the circuit is,

$$C = S \cdot A + (1 - S) \cdot (1 - B). \quad (7)$$

If we assume A, B, C, and S are stochastic bit streams, and P_A, P_B, P_C, and P_S stand for their corresponding probabilities, we will get

$$P_C = P_S \cdot P_A + (1 - P_S) \cdot (1 - P_B). \quad (8)$$

We define a, b, and c as the corresponding bipolar coding format of P_A, P_B, and P_C, i.e., $a = 2P_A - 1$, $b = 2P_B - 1$, $c = 2P_C - 1$. If we set $P_S = 0.5$, equation (8) can be rewritten as follows,

$$c = 0.5 \cdot (a - b). \quad (9)$$

3.2.3 Multiplication

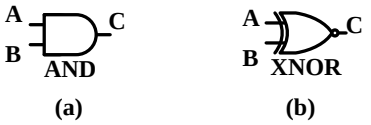


Figure 1: Multiplication in stochastic computing.

Multiplication can be implemented with a two-input AND gate as shown in Fig. 1(a) for the unipolar coding format, and with a two-input XNOR gate as shown in Fig. 1(b) for the bipolar coding format. Multiplication based on the AND gate for the unipolar coding format is straightforward. We will explain multiplication based on the XNOR gate for the bipolar coding format as follows. In Fig. 1(b), we have

$$C = A \cdot B + (1 - A) \cdot (1 - B). \quad (10)$$

If we define P_A, P_B, and P_C as the probabilities of the streams A, B, and C, and a, b, and c as the corresponding bipolar coding format of P_A, P_B, and P_C (i.e., $a = 2P_A - 1$, $b = 2P_B - 1$, and $c = 2P_C - 1$), we can rewrite (10) as follows,

$$\frac{c+1}{2} = \frac{a+1}{2} \cdot \frac{b+1}{2}$$

$$+ (1 - \frac{a+1}{2}) \cdot (1 - \frac{b+1}{2}). \quad (11)$$

From (11), we have

$$c = a \cdot b. \quad (12)$$

3.3 The FSM-Based SCEs

In addition to the combinational logic-based SCEs, FSM-based SCEs can be used to perform some sophisticated functions more efficiently, such as exponential and tanh functions [3, 9]. These FSM-based SCEs can be developed based on the linear state transition diagram shown in Fig. 2. Although several FSM-based SCEs have been developed so far [3, 9], here we demonstrate the CUDA implementations of only two FSM-based SCEs. The others can be implemented in a similar way.

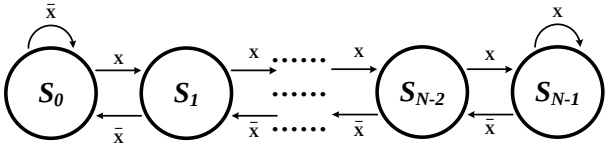


Figure 2: A linear state transition diagram (the output Y, which is not shown in the figure, is determined only by the current state).

3.3.1 Compute absolute value stochastically

Based on the state transition diagram shown in Fig. 2, if we set the output $Y = 1$ for the state S_i which satisfies:

- (1) $0 \le i < N/2$ and i is even,
- (2) $N/2 \le i \le N - 1$ and i is odd,

the FSM will perform the function to compute the absolute value stochastically with the bipolar coding format for both input and output (i.e., $y = 2P_Y - 1$ and $x = 2P_X - 1$) [10]:

$$y = |x|. \quad (13)$$

A straightforward way to implement the FSM in CUDA is to use multiple branches, such as $\{switch \cdots case \cdots\}$. However, this approach will generate a lot of divergences, which will cause the CUDA performance to suffer because all the paths of a branch are executed in a single SM [1]. To get better performance, we should avoid divergences as much as we can. Our solution to the divergence issue is to synthesize the FSM at the circuit level, which directly uses the boolean expressions (instead of the multiple branches) to implement the function of the FSM. For simplicity, our implementation is based on a four-state FSM. Fig. 3 shows the corresponding circuit diagram for the four-state FSM including two D-flip-flops and two combinational circuits with input X and ouput Y. The boolean expressions for inputs of two D-flip-flops and output Y are also described in the figure. By circuit-level synthesis of the FSM, we can simplify the CUDA simulation code.

3.3.2 The stochastic tanh function

Based on the state transition diagram shown in Fig. 2, if we set the output $Y = 1$ for the state S_i which satisfies $N/2 \le i \le N - 1$, the approximate transfer function will be

$$y = \frac{e^{\frac{N}{2}x} - e^{-\frac{N}{2}x}}{e^{\frac{N}{2}x} + e^{-\frac{N}{2}x}}, \quad (14)$$

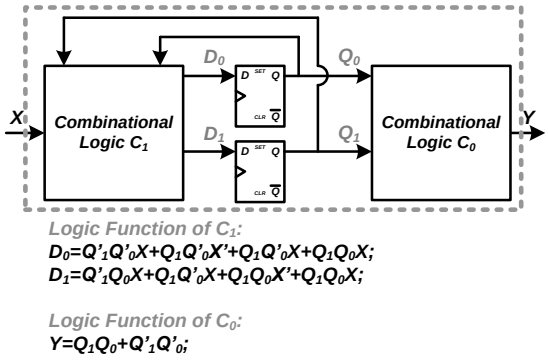


Figure 3: The circuit digram based on a four-state FSM for stochastic absolute value function.

where x is the bipolar coding of P_X and y is the bipolar coding of P_Y. The details of this FSM-based SCEs can be found in [3]. Similar to the stochastic absolute value function, we use another four-state FSM to implement the stochastic tanh function. The only difference is the boolean expression for the output Y. The detailed description is omitted here due to space restrictions.

3.4 CUDA Implementation and Device Functions

In our GPU-based simulation, all SCEs have been implemented as device functions. Each device function handles one stochastic bit stream. All these device functions can be easily customized to implement a stochastic algorithm. In this sesction, we use the stochastic absolute value function as an example to present the implementation of a device function for stochastic simulation and post the detailed code at http://www.arctic.umn.edu/gpu-simulation.tgz.

As shown in Fig. 4, the device function *SCAbs* takes a biploar stochastic bit stream *StreamIn* as input and stores a uniploar bit stream for the corresponding absolute value back to *StreamOut*. Inside the device function, for each bit in a stochastic stream, function *FSMABS* is called to get the next state and a output bit is filled into the output stream. The implementaion of function *FSMABS* is based on the four-state FSM described in Section 3.3.1.

4. A CASE STUDY FOR GPU-BASED SIMULATION

In this section, we use the stochastic implementation of the frame difference-based image segmentation algorithm [7] as a case study to demonstrate the GPU-based parallel simulation approach for a stochastic computing system. Although the frame difference-based image segmentation algorithm is very simple, more complex stochastic computing systems can be simulated on the GPU in a similar way.

The frame difference-based image segmentation algorithm uses a pixel's value at the current frame minus the pixel's value in the same location of the last frame to check if the absolute value of the difference is greater than a threshold. If yes, we say the pixel at the current frame belongs to the foreground, otherwise, it belongs to the background.

```
__device__ void FSMABS(char array[2], int x,
char currentstate) {
    char curQ1, curQ0, newQ1, newQ0;

    //array[1] stands for the next state
    //array[0] stands for the output

    curQ1 = currentstate / 2;
    curQ0 = currentstate % 2;

    newQ1 = ((!curQ1) && curQ0 && x) ||
            ((curQ1) && (!curQ0) && x) ||
            ((curQ1) && (curQ0) && (!x)) ||
            ((curQ1) && (curQ0) && x);

    newQ0 = ((!curQ1) && (!curQ0) && x) ||
            ((curQ1) && (!curQ0) && (!x)) ||
            ((curQ1) && (!curQ0) && (x)) ||
            ((curQ1) && (curQ0) && x);

    array[0] = !(newQ1 ^ newQ0);
    array[1] = 2 * newQ1 + newQ0; }

/*=============================================*/

__device__ void SCAbs(char StreamOut[],
char StreamIn[]) {
    int i;
    char tempArray[2];
    char currentstate;

    FSMABS(tempArray, StreamIn[0], 0);
    StreamOut[0] = tempArray[0];
    char nextstate = tempArray[1];

    for (i = 1; i < BITLENGTH; i++) {
        currentstate = nextstate;

        FSMABS(tempArray, StreamIn[i],
               currentstate);

        StreamOut[i] = tempArray[0];
        nextstate = tempArray[1]; } }
```

Figure 4: CUDA implementation of the stochastic absolute value function.

Based on the CUDA implmentations of the basic SCEs, we develop a CUDA kernel function *SCfdKernel* for the frame difference-based image segmentation algorithm. In the kernel function, the segmentation result of each pixel is performed by a single CUDA thread, which includes the following steps:

- Convert a pixel value at the current frame (i.e., X_t) into a stochastic bit stream by calling the *D2S* device function.
- Convert a pixel value in the same location of the last frame (i.e., X_{t-1}) into a stochastic bit stream by calling the *D2S* device function again.
- Compute $(X_t - X_{t-1})$ stochastically by calling the scaled subtraction function. P_S is a predefined stochastic bit stream with probability 0.5.
- Compute $|X_t - X_{t-1}|$ by calling the stochastic absolute value function.
- Compute $(|X_t - X_{t-1}| - Th)$ stochastically by calling the scaled subtraction function again. *Threshold* is a predefined stochastic bit stream with probability $\frac{Th}{256}$.

- Convert the result of $(|X_t - X_{t-1}| - Th)$ to either 0 (foreground) or 1 (background) by calling the stochastic tanh function.
- Convert the processed stream into a deterministic value to be displayed.

5. EXPERIMENTAL RESULTS

To validate the GPU-based simulation for stochastic computing, we have conducted three experiments. These experiments are run on a Linux machine with the CPU specification shown in Table 2 and the GPU specification shown in Table 3. Each of the three experiments has a CPU-based C code version and a GPU-based CUDA C code version. Both versions of the codes are well optimized and perform exactly the same functions. The first experiment evaluates the execution time of each of the basic functions introduced in Section 3 based on the two different simulation platforms. The second and the third experiments use the stochastic implementation of the frame difference-based image segmentation algorithm introduced in Section 4 as a baseline system of stochastic computing. The second experiment evaluates the execution time of the entire system based on six different lengths of the stochastic bit streams. The third experiment evaluates the execution time of the entire system based on five different image resolutions.

5.1 Simulation Performance Comparison of the SCEs

In this experiment, we measure the execution time for each of the basic functions in stochastic computing based on six different lengths of the stochastic bit stream on both the GPU and the CPU, and compare the performance. We repeat each test 1000 times. Since there was only the single application running on the systems being measured, the measurement variance in all of our experiments was less than 1%. Consequently, we show only the average values in our performance results.

Table 4 shows the results for the *D2S* function, from which the GPU-based simulation achieves up to 165 times performance speedup compared to the CPU-based simulation. Table 5 shows the results for the combinational logic-based SCEs, from which the GPU-based simulation achieves up to 90 times performance speedup compared to the CPU-based simulation. Table 6 shows the results for the FSM-based SCEs, from which the GPU-based simulation achieves up to 27 times performance speedup compared to the CPU-based simulation.

These results are below our expectations given by the high performance parallel computing capability of the GPUs. Another observation from the measured results is that the performance speedup increases as the bit length of a stochastic stream increases except for the case of the FSM-based SCEs. This is not a surprise because GPU computing will show significant benefits to solve a large-scale problem in contrast to the multicore CPU. For the FSM-based SCEs, the speedup does not change too much when the bit length increases. The possible reason is that four-state FSM in our implementation is not sensitive to the changes of the bit length.

It should be noted here that our CPU implementation is a single-threaded application on the host. If we develop a multithreaded program to implement the basic SCEs, the CPU implementation could have much better performance than our current implementation. Multicore and multithreaded CPUs could also be helpful for stochastic computing. However, the number of cores of a typical CPU is much less than the number of cores in a GPU. We expect the GPUs can still have good performance speedups compared to multithreaded CPU implementations because the GPUs can acheive massive parallelism for the applications without any data dependency. For stochastic simulation, each of bit stream is indepedent of the other streams, which fits GPU computing very well. It would be very interesting to compare openMP/MPI implementations with CUDA implementations for stochastic simulation. We leave this comparison for our future work.

Table 4: Simulation performance comparison for the *D2S* function.

Bit Length	GPU Run Time (s)	CPU Run Time (s)	Speedup
256	0.001	0.1	100
512	0.002	0.2	100
1024	0.003	0.41	136.67
2048	0.005	0.82	164
4096	0.01	1.65	165
8192	0.02	3.29	150

Table 5: Simulation performance comparison for the combinational logic-based SCEs.

Bit Length	GPU Run Time (s)	CPU Run Time (s)	Speedup
256	0.003	0.03	10
512	0.005	0.07	14
1024	0.007	0.14	20
2048	0.01	0.29	36.3
4096	0.01	0.59	59
8192	0.01	1.18	90.8

Table 6: Simulation performance comparison for the FSM-based SCEs

Bit Length	GPU Run Time (s)	CPU Run Time (s)	Speedup
256	0.004	0.11	27.5
512	0.008	0.22	27.5
1024	0.017	0.45	26.47
2048	0.03	0.91	26.8
4096	0.07	1.84	27.1
8192	0.14	3.69	26.9

5.2 Simulation Performance Comparison of the Entire System

We conduct the second experiment to measure the overall simulation times of the stochastic implementation of the

frame difference-based image segmentation algorithm using six different lengths of the stochastic bit streams. As shown in Table 7, the simulations using the GPU have much shorter simulation times than the ones based on the CPU, with up to 98 times performance speedup. It should be noted here that our current CUDA code only implements pixel-level parallelism. To further reduce the simulation time, we can exploit the parallelism at the bit-level, i.e., use multiple threads to perform the function of a single SCE simultaneously.

In the last experiment, we measure the execution time for five different image resolutions as shown in Table 8. The GPU-based simulation still has significant performance speedup. When the image resolution is increased, we have more threads available in the system. Thus, a higher thread level parallelism can further improve the performance. It can be seen that the speedups in Table 8 are much higher than the results in Table 7.

Table 7: Simulation performance comparison of the stochastic implementation of the frame difference-based image segmentation algorithm for six different lengths of stochastic bit streams (image resolution: 176 × 144).

Bit Length	GPU Run Time (s)	CPU Run Time (s)	Speedup
256	0.01673	0.3	17.63
512	0.01909	1.59	31.01
1024	0.02431	1.17	48.21
2048	0.03447	2.35	68.06
4096	0.0553	4.72	85.37
8192	0.09609	9.45	98.37

Table 8: Performance comparison of the stochastic implementation of the frame difference-based image segmentation algorithm for five different image resolutions (bit length: 8192).

Resolution	GPU Run Time (s)	CPU Run Time (s)	Speedup
176 × 144	0.09609	9.45	98.37
352 × 288	0.3509	37.14	105.84
640 × 480	1.012	112.55	111.22
1280 × 720	2.880	337.82	117.42
1920 × 1080	6.33	753.83	119.13

5.3 Performance Considerations and CUDA Optimizations

Proper thread block size and kernel configuration are very important for achieving high performance for a CUDA application. In our experiments, we measured the results for three different thread block size: 8x8,16x16,and 32x32. Table 9 shows the results for six different lengths of stochastic bit streams when image resolution is 176x144. It can be seen from the table that 16x16 thread block has the best performance for the frame difference algorithm. These results are consistent with our analysis and profiling. When the thread block size is 8x8, totally only 512 threads is assigned in a SM for scheduling. The occupany is 0.333 and thread-level parallelism is lost because there are not enough threads in the SM for scheduling and memory latency cannot be hidden by overlapping. On the other hand, when the thread block is 32x32, one SM allows only one thread block to be scheduled. The occupancy is 0.667 because the block size is too large and some blocks have been lost. The best configuration is 16x16, in which 5 blocks are brought into a SM and occupancy is 0.833. In general, the higher occupancy a kernel has, the better performance it achieves if we do not consider the limitations of register and shared memory usage. For our GPU-based simulation, we do not use any shared memory and register usage is not a bottlebeck. Therefore, 16x16 is the optimum thread block size for the best performance.

Table 9: Performance comparison of the stochastic implementation of the frame difference-based image segmentation algorithm for three thread block sizes (image resoluation: 176x144).

Bit Length	GPU Run Time with 8x8(s)	GPU Run Time with 16x16(s)	GPU Run Time with 32x32(s)
256	0.01738	0.01673	0.0169
512	0.02042	0.01909	0.01953
1024	0.02717	0.02431	0.02555
2048	0.04009	0.03447	0.03688
4096	0.06714	0.0553	0.06003
8192	0.12037	0.09609	0.10539
Occupancy	0.333	0.833	0.667

In addition to proper kernel configurartion, another performance consideration for CUDA optimization is the L1 cache size configuration. For Fermi GPUs, both the shared memory and the L1 data cache share the same on-chip memory. The total size is 64KB. CUDA programmer can use either 16KB or 48KB L1 cache. The default L1 cache size is 16KB. This size of L1 cache can be changed to 48KB by calling function cudaFuncSetCacheConfig. In our kernel function *ScfdKernel*, each CUDA thread processes one image pixel as described in Section 4, we define two local arrays to store stochastic bit streams: one is for the input, the other for output. These two arrays are allocated in local memory. According to the CUDA programming guide, both global and local memories are cached for Fermi GPUs. If the size of the L1 data cache can be increased, we expect to get a higher cache hit ratio for the L1 cache, which can improve the performance of a CUDA program. In our GPU-based simulation, we do not use shared memory for CUDA implementations. By adjusting the size of the L1 cache to 48KB, we tested the system performance for six different bit lengths. As shown in Table 10, the 48KB L1 cache can have much better performance than the 16KB L1 cache up to 32% speedup. These results are expected because the large L1 cache size will increase the hit ratio for local memory accesses, which benefits the system performance.

6. CONCLUSIONS

In this paper, we propose a GPU-based parallel computing approach to speed up the simulation of a stochastic computing system. We demonstrate the GPU (CUDA) implementa-

Table 10: Performance comparison of the stochastic implementation of the frame difference-based image segmentation algorithm for two different L1 cache sizes (image resoluation: 176x144).

Bit Length	GPU Run Time with 48KB L1(s)	GPU Run Time with 16KB L1(s)	Speedup (ptg)
256	0.01132	0.01673	32.34%
512	0.01383	0.01909	27.55%
1024	0.01961	0.02431	19.33%
2048	0.03072	0.03447	10.88%
4096	0.05298	0.0553	4.20%
8192	0.09608	0.09609	0.01%

tions of the basic computational elements in stochastic computing, and use the frame difference-based image segmentation algorithm as a case study to validate our approach. Measured results show that our GPU-based simulation can achieve up to 119 times performance speedup compared to the CPU-based simulation for large images.

We note that currently the parallel computing is performed at the pixel level, i.e., multiple pixels are processed simultaneously. In fact, we can further speed up the simulation of stochastic computing by performing the parallel computing at the bit level, i.e., we can use multiple threads to simulate a single SCE simultaneously. Although the combinational logic-based SCEs can be easily simulated in parallel at the bit level, the FSM-based SCEs and the *D2S* function are hard to simulate in parallel at the bit level. Future work will focus on how to simulate each SCE in parallel at the bit level, especially for the FSM-based SCEs and the *D2S* function.

7. ACKNOWLEDGMENT

This paper is based upon work supported by the US National Science Foundation (NSF) under Grant ITR-0937060 to the Computing Research Association for the CIFellows Project and by NSF grant no. EAR-0941666, the Minnesota Supercomputing Institute, and NVIDIA Corporation. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF.

8. REFERENCES

[1] CUDA Toolkit 4.0 CURAND Guide. *NVIDIA Corporation*, version 12.3, January 2011.

[2] P. Bailey, J. Myre, S. D. C. Walsh, D. J. Lilja, and M. O. Saar. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In *International Conference on Parallel Processing, ICPP'09*, 2009.

[3] B. D. Brown and H. C. Card. Stochastic neural computation I: Computational elements. *IEEE Transactions on Computers*, 50(9):891–905, September 2001.

[4] B. D. Brown and H. C. Card. Stochastic neural computation II: Soft competitive learning. *IEEE Transactions on Computers*, 50(9):906–920, September 2001.

[5] A. Elgammal, D. Harwood, and L. Davis. Non-parametric model for background subtraction. In *FRAME-RATE WORKSHOP, IEEE*, pages 751–767, 2000.

[6] B. R. Gaines. Stochastic computing systems. *Advances in Information System Science, Plenum*, 2(2):37–172, 1969.

[7] R. C. Gonzalez and R. E. Woods. Digital image processing, 3rd edition. *Prentice Hall*, 2008.

[8] M. A. Lastras-Montano, M. M. Michael, and J. A. Bivens. Dynamic work scheduling for GPU systems. In *International Workshop on GPUs and Scientific Applications, GPUScA'10*, 2010.

[9] P. Li and D. J. Lilja. A low power fault-tolerance architecture for the kernel density estimation based image segmentation algorithm. In *IEEE International Conference on Application - specific Systems, Architectures and Processors, ASAP'11*, 2011.

[10] P. Li and D. J. Lilja. Using stochastic computing to implement digital image processing algorithms. In *IEEE International Conference on Computer Design, ICCD'11*, 2011.

[11] A. Monitzer. Fluid simulation with CUDA using the lattice boltzmann method. In *International Workshop on GPUs and Scientific Applications, GPUScA'10*, 2010.

[12] J. Myre, S. Walsh, D. J. Lilja, and M. O. Saar. Performance analysis of single-phase, multiphase, and multicomponent lattice-Boltzmann fluid flow simulations on GPU clusters. In *Concurrency and Computation: Practice and Experience*, (to appear).

[13] L. Perrotte and G. Saupin. Fast GPU perspective grid construction and triangle tracing for exhaustive ray tracing of highly coherent rays. In *International Workshop on GPUs and Scientific Applications, GPUScA'10*, 2010.

[14] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja. An architecture for fault-tolerant computation with stochastic logic. *IEEE Transactions on Computers*, 60(1):93–105, January 2010.

[15] W. Qian and M. D. Riedel. The synthesis of robust polynomial arithmetic with stochastic logic. In *45th ACM/IEEE Design Automation Conference, DAC'08*, pages 648–653, 2008.

[16] G. Vigueras, J. M. Orduna, M. Lozano, J. M. Cecilia, and J. M. Garcia. Improving the GPU-based collision check procedure for distributed crowd simulations. In *International Workshop on GPUs and Scientific Applications, GPUScA'10*, 2010.

[17] S. D. Walsh, M. O. Saar, P. Bailey, and D. J. Lilja. Accelerating geo-science and engineering system simulations on graphics hardware. *Journal of Computers and Geosciences*, 35(12):2353–2364, December 2009.

ForOpenCL: Transformations Exploiting Array Syntax in Fortran for Accelerator Programming

Matthew J. Sottile
Galois, Inc.
Portland, OR 97204
mjsottile@computer.org

Craig E Rasmussen
Los Alamos National Laboratory
Los Alamos, NM 87545
crasmussen@lanl.gov

Wayne N. Weseloh
Los Alamos National Laboratory
weseloh@lanl.gov

Robert W. Robey
Los Alamos National Laboratory
brobey@lanl.gov

Daniel Quinlan
Lawrence Livermore National Laboratory
dquinlan@llnl.gov

Jeffrey Overbey
University of Illinois at Urbanna-Champaign
overbey2@illinois.edu

ABSTRACT

Emerging GPU architectures for high performance computing are well suited to a data-parallel programming model. This paper presents preliminary work examining a programming methodology that provides Fortran programmers with access to these emerging systems. We use array constructs in Fortran to show how this infrequently exploited, standardized language feature is easily transformed to lower-level accelerator code. The transformations in ForOpenCL are based on a simple mapping from Fortran to OpenCL. We demonstrate, using a stencil code solving the shallow-water fluid equations, that the performance of the ForOpenCL compiler-generated transformations is comparable with that of hand-optimized OpenCL code.

1. INTRODUCTION

This paper presents a compiler-level approach for targeting a single program to multiple, and possibly fundamentally different, processor architectures. This technique allows the application programmer to adopt a single, high-level programming model without sacrificing performance. We suggest that existing data-parallel features in Fortran are well-suited to applying automatic transformations that generate code specifically tuned for different hardware architectures using low-level programming models such as OpenCL. For algorithms that can be easily expressed in terms of whole array, data-parallel operations, writing code in Fortran and transforming it automatically to specific low-level implementations removes the burden of creating and maintaining multiple versions of architecture specific code.

The peak performance of these newer accelerator architectures can be substantial. Intel expects a teraflop for the SGEMM benchmark with their Knights Ferry processor while the performance of the M2090 NVIDIA Tesla processor is in the same neighborhood [7]. Unfortunately the performance that many of the new accelerator architectures offer comes at a cost. Architectural changes are trending toward multiple heterogeneous cores and less of a reliance on superscalar instruction level parallelism and hardware managed memory hierarchies (such as traditional caches).

These changes place a heavy burden on application programmers as they work to adapt to these new systems. An especially challenging problem is not only how to program to these new architectures — considering the massive scale of concurrency available — but also how to design programs that are portable across the changing landscape of computer architectures. How does a programmer write one program that can perform well on both a conventional multicore CPU *and* a GPU (or any other emerging many-core architectures)?

A directive-based approach, such as OpenMP or the Accelerator programming model from the Portland Group [14], is one solution to this problem. However, in this paper we take a somewhat different approach. A common theme amongst the new processors is the emphasis on data-parallel programming. This model is well-suited to architectures that are based on either vector processing or massively parallel collections of simple cores. The recent CUDA and OpenCL programming languages are intended to support this programming model.

The problem with OpenCL and CUDA is that they expose too much detail about the machine architecture to the programmer [15]. The programmer is responsible for explicitly managing memory (including the staging of data back and forth between the host CPU and the accelerator device) and specifically taking into account architectural differences (such as whether the architecture contains vector units). While these languages have been attractive as a method for early adopters to utilize these new architectures, they are less attractive to programmers who do not have the time or resources to manually port their code to every new architecture and programming model that emerges.

1.1 Approach

We demonstrate that a subset of Fortran map surprisingly well onto GPUs when transformed to OpenCL kernels. This data-parallel subset includes: array syntax using assignment statements and binary operators, array constructs like `WHERE`, and the use of pure and elemental functions. In addition, we provide new functions that explicitly take advantage of the stencil geometry of the problem domain we consider. Note that this subset of the Fortran language is implicitly parallel. This programming model *does not require explicit declaration of parallelism within the program.* In addition, programs are expressed using entirely standard Fortran so it can be compiled for and executed on a single core without concurrency.

Transformations are supplied that provide a mechanism for converting Fortran procedures written in the Fortran subset described in this paper to OpenCL kernels. We use the ROSE compiler infrastructure[1] to develop these transformations. ROSE uses the Open Fortran Parser[2] to parse Fortran 2008 syntax and can generate C-based OpenCL. Since ROSE's intermediate representation (IR) was constructed to represent multiple languages, it is relatively straightforward to transform high-level Fortran IR nodes to C OpenCL nodes. This work is also applicable to transformations to vendor-specific languages, similar to OpenCL, such as the NVIDIA CUDA language.

Transformations for arbitrary Fortran procedures are not attempted. Furthermore, a mechanism to transform the calling site to automatically invoke OpenCL kernels is not provided at this time. While it is possible to accomplish this task within ROSE, it is considered outside the scope of this paper. However, ForOpenCL provides via Fortran interfaces a mechanism to call the C OpenCL runtime and enable Fortran programmers to access OpenCL kernels generated by the supplied transformations.

We study the automatic transformations for an application example that is typical of stencil codes that update array elements according to a fixed pattern. Stencil codes are often employed in applications based on finite-difference or finite-volume methods in computational fluid dynamics (CFD). The example described later in this paper is a simple shallow-water model in two dimensions using finite volume methods. Stencil-like patterns appear in a number of other contexts as well. In image processing, they appear in convolution-based algorithms in which small kernels are convolved with an image to implement denoising, edge detection, and other common operators. Similar stencil operators appear in general signal processing applications as well.

Finally, we examine the performance of the Fortran data-parallel abstraction when transformed to OpenCL to run on GPU architectures. The performance of automatically transformed code is compared with a hand-optimized OpenCL version of the shallow-water code.

We do not perform any additional analysis of the code to identify parallelism beyond that present in the data parallel operations that we focus on in this paper. Additional program analysis methods may be investigated to study their applicability in future versions of this work.

[1] `http://www.rosecompiler.org/`
[2] `http://fortran-parser.sf.net/`

2. PROGRAMMING MODEL

A question that one may pose is "*Why choose Fortran and not a more modern language like X for programming accelerator architectures?*" The recent rise in interest in concurrency and parallelism at the language level due to multicore CPUs and many-core accelerators has driven a number of new language developments, both as novel languages and extensions on existing ones. However, for many scientific users with existing codes written in Fortran, new languages and language extensions to use novel new architectures present a challenge: how do programmers effectively use them while avoiding rewriting code and potentially growing dependent on a transient technology that will vanish tomorrow? In this paper we explore the constructs in Fortran that are particularly relevant to GPU architectures.

In this section we present the Fortran subset employed in this paper. This sub-setting language will allow scientific programmers to stay within the Fortran language and yet have direct access to GPU hardware. We start by examining how this programming model relates to developments in other languages.

2.1 Comparison to Prior Fortran Work

A number of previous efforts have exploited data-parallel programming at the language level to utilize novel architectures. The origin of the array syntax adopted by Fortran in the 1990 standard can be found in the APL language [6]. These additions to Fortran allowed parallelism to be expressed with whole-array operations at the expression level, instead of via parallelism within explicit DO-loops, as implemented in earlier variants of the language (e.g., IVTRAN for the Illiac IV).

The High Performance Fortran (HPF) extension of Fortran was proposed to add features to the language that would enhance the ability of compilers to emit fast parallel code for distributed and shared memory parallel computers [10]. One of the notable additions to the language in HPF was syntax to specify the distribution of data structures amongst a set of parallel processors. HPF also introduced an alternative looping construct to the traditional DO-loop called `FORALL` that was better suited for parallel compilation. An additional keyword, `INDEPENDENT`, was added to allow the programmer to indicate when the loop contained no loop-order dependencies that allowed for parallel execution. These constructs are similar to `DO CONCURRENT`, an addition to Fortran in 2008.

Interestingly, the parallelism features introduced in HPF did not exploit the new array features introduced in 1990 in any significant way, relying instead on explicit loop-based parallelism. This restriction allowed the language to support parallel programming that wasn't easily mapped onto a pure data-parallel model. The `SHADOW` directive introduced in HPF-2, and the `HALO` in HPF+ [1] bear some similarity to the halo region concept that we discuss in this paper.

In some instances though, a purely data-parallel model is ap-

propriate for part or all of the major computations within a program. One of the systems where programmers relied heavily on higher level operations instead of explicit looping constructs was the Thinking Machines Connection Machine 5 (CM-5). A common programming pattern used on the CM-5 (that we exploit in this paper) was to write whole-array operations from a global perspective in which computations are expressed in terms of operations over the entire array instead of a single local index. The use of the array shift intrinsic functions (like `CSHIFT`) were used to build computations in which arrays were combined by shifting the entire arrays instead of working on local offsets based on single indices. A simple 1D example is one in which an element is replaced with the average of its own value and that of its two direct neighbors. Ignoring boundary indices that wrap around, explicit indexing will result in a loop such as:

```
do i = 2,(n-1)
  Xnew(i) = (X(i-1) + X(i) + X(i+1)) / 3
end do
PTR_SWAP(Xnew, X, tmp_ptr)
```

In this loop, it is necessary to manually implement a double buffering scheme in order to avoid mixing values computed in the current execution of the loop with values from a prior execution of the loop. When shifts are employed, this can be expressed as:

```
X = (cshift(X,-1) + X + cshift(X,1)) / 3
```

The semantics of the shift intrinsic and operators like `+` applied to the whole array make it unnecessary to manually implement the double buffering scheme found in the loop above. Similar whole array shifting was used in higher dimensions for finite difference codes within the computational physics community for codes targeting the CM-5 system. Research in compilation of stencil-based codes that use shift operators targeting these systems is related to the work presented here [3].

The whole-array model was attractive because it deferred responsibility for optimally implementing the computations to the compiler. Instead of relying on a compiler to infer parallelism from a set of explicit loops, the choice for how to implement loops was left entirely up to the tool.

Unfortunately, this had two side effects that have limited broad acceptance of the whole-array programming model in Fortran. First, programmers must translate their algorithms into a set of global operations. Finite difference stencils and similar computations are traditionally defined in terms of offsets from some central index. Shifting, while conceptually analogous, can be awkward to think about for high dimensional stencils with many points. Second, the semantics of these operations are such that all elements of an array operation are updated as if they were updated simultaneously. In a program where the programmer explicitly manages arrays and loops, double buffering techniques and user managed temporaries are used to maintain these semantics. Limited attention to optimizing memory usage due to this intermediate storage by compilers has led to these constructs seeing little adoption by programmers.

An interesting line of language research that grew out of HPF was that associated with the ZPL language at the University of Washington [5] and Chapel, an HPCS language developed by Cray [4]. In ZPL, programmers adopt a similar global view of computation over arrays, but define their computations based on regions, which provide a local view of the set of indices that participate in the update of each element of an array. A similar line of research in the functional language community has investigated array abstractions for expressing whole-array operations in the Haskell language in the REPA (regular, shape-polymorphic, parallel array) library [8].

2.2 Fortran Language Subset

The static analysis and source-to-source transformations used in this work require the programmer to use a language subset that employs a data-parallel programming model. In particular, it encourages the use of array notation, pure elemental functions, and pure procedures. From these language constructs, we are able to easily transform Fortran procedures to a lower-level OpenCL kernel implementation.

Array notation

Fortran has a rich array syntax that allows programmers to write statements in terms of whole arrays or subarrays, with data-parallel operators to compute on the arrays. Array variables can be used in expressions based on whole-array operations. For example, if `A`, `B`, and `C` are all arrays of the same rank and shape and `s` is a scalar, then the statement

```
C = A + s*B
```

results in the element-wise sum of `A` and the product of `s` times the elements of `B` being stored in the corresponding elements of `C`. The first element of `C` will contain the value of the first element of `A` added to the first element of `c*B`. Note that no explicit iteration over array indices is needed and that the individual operators, plus, times, and assignment are applied by the compiler to individual elements of the arrays independently. Thus the compiler is able to spread the computation in the example across any hardware threads under its control.

Pure elemental functions

An elemental function consumes and produces scalar values, but can be applied to variables of array type such that the function is applied to each and every element of the array. This allows programmers to avoid explicit looping and instead simply state that they intend a function to be applied to every element of an array in parallel, deferring the choice of implementation technique to the compiler. Pure elemental functions are intended to be used for data-parallel programming, and as a result must be side effect free and mandate an `intent(in)` attribute for all arguments.

For example, the basic array operation shown above could be refactored into a pure elemental function,

```
pure elemental real function foo(a, b, s)
  real, intent(in) :: a, b, s
  foo = a + s*b
end function
```

and called with

```
C = foo(A, B, s)
```

Note that while `foo` is defined in terms of purely scalar quantities, it can be *applied* to arrays as shown. While this may seem like a trivial example, such simple functions may be composed with other elemental functions to perform powerful computations, especially when applied to arrays. Our prototype tool transforms pure elemental functions to inline OpenCL functions. Thus there is no penalty for usage of pure elemental functions and they provide a convenient mechanism to express algorithms in simpler segments.

It should be noted that in Fortran 2008, the concept of impure elemental functions was introduced. This requires that elemental functions now need to be explicitly labeled as pure to indicate that they are side effect free.

Pure procedures

Pure procedures, like pure elemental functions, must be free of side effects. Unlike pure elemental functions that require arguments to have an `intent(in)` attribute, they may change the contents of array arguments that are passed to them. The absence of side effects removes ordering constraints that could restrict the freedom of the compiler to invoke pure functions out of order and possibly in parallel. Procedures and functions of this sort are also common in pure functional languages like Haskell, and are exploited by compilers in order to emit parallel code automatically due to their suitability for compiler-level analysis.

Since pure procedures don't have side effects they are candidates for running on accelerators in OpenCL. Currently our prototype tool only transforms pure procedures to OpenCL kernels that *do not* call other procedures, except for pure elemental functions, either defined by the user or intrinsic to Fortran.

2.3 New Procedures

Borrowing ideas from ZPL, we introduce a concept of a region to Fortran with a set of functions that allow programmers to work with subarrays in expressions. In Fortran, these functions return a copy of or a pointer to an existing array or array section. This is unlike ZPL, where regions are analogous to index sets and are used primarily for address resolution within an array without dictating storage related behavior. The functions that we propose are similar in that they allow a programmer to deal with index regions that are meaningful to their algorithm, and automatically induce a halo (or ghost) cell pattern as needed in the implementation generated by the compiler, where the size of an array is implicitly increased to provide extra array elements surrounding the interior portion of the array. It is important to note, however, that all memory allocated by the programmer must explicitly contain the extra array elements in the halo.

Region functions are similar to the shift operator as they can be used to reference portions of the array that are shifted with respect to the interior portion. However, unlike the shift operator, regions are not expressed in terms of boundary conditions and thus don't explicitly *require* a knowledge of, nor the application of, boundary conditions locally (global boundary conditions must be explicitly provided by the programmer outside of calls to kernel procedures). Thus, as will be shown below, regions are more suitable for usage by OpenCL thread groups which access only local subsections of an array stored in global memory.

ForOpenCL provides two new functions that are defined in Fortran and are used in array-syntax operations. Each function takes an integer array halo argument that specifies the number of ghost cells on either side of a region, for each dimension. For example `halo = [left, right, down, up]` specifies a halo for a two-dimensional region. These functions are:

- `region_cpy(array, halo)`: a pure function that returns a copy of the interior portion of the array specified by halo.
- `region_ptr(array, halo)`: an impure function that returns a pointer to the portion of the array specified by halo.

It should be noted that the function `region_cpy` is pure and thus can be called from within a pure kernel procedure, and `region_ptr` is impure because it aliases the array parameter. However as will be shown below, the usage of `region_ptr` is constrained so that it does not introduce side effects in the functions that call it. These two functions are part of the language recognized by the compiler and though `region_cpy` returns a copy of a portion of an array *semantically*, the compiler is not forced to actually make a copy and is free to enforce copy semantics through other means. In addition to these two new functions, ForOpenCL provides the compiler directive, `$OFP PURE, KERNEL`, which specifies that a pure subroutine can be transformed to an OpenCL kernel and that the subroutine is pure except for calls to `region_ptr`. These directives are not strictly necessary for the technique described in this paper, but aid in automated identification of specific kernels to be transformed to OpenCL. A directive-free implementation would require the transformation tool be provided the set of kernels to work via a user defined list.

2.4 Advantages

There are several advantages to this style of programming using array syntax, regions, and pure and elemental functions:

- There are no loops or index variables to keep track of. Off by one index errors and improper handling of array boundaries are a common programming mistake.
- The written code is closer to the algorithm, easier to understand, and is usually substantially shorter.
- Semantically the intrinsic function `region_cpy` returns an array by value. This is usually what the algorithm requires.

- Pure elemental functions are free from side effects, so it is easier for a compiler to schedule the work to be done in parallel.

Data parallelism has been called collection-oriented programming by Blelloch [2]. As the `cshift` function and the array-valued expressions all semantically return a value, this style of programming is also similar to functional programming (or value-oriented programming). It should be noted that the sub-setting language we employ goes beyond pure data parallelism by the use of pure (other than calls to `region_ptr`) subroutines and not just elemental functions.

Unfortunately, this style of programming has never really caught on because when array syntax was first introduced in Fortran, performance of codes using these features was relatively poor and thus programmers shied away from using array syntax (even recently, some are actively counseling against its usage because of performance issues [12]). Thus the Fortran community was caught in a classic "chicken-and-egg" conundrum: (1) programmers didn't use it because it was slow; and (2) compilers vendors didn't improve it because programmers didn't use it. A goal of this paper is to demonstrate that parallel programs written in this style of Fortran can achieve good performance on accelerator architectures.

2.5 Restrictions

Only pure Fortran procedures are transformed into OpenCL kernels. This restriction is lifted slightly to allow calls to `region_ptr` from within a kernel procedure. The programmer must explicitly call these kernels using Fortran interfaces in the ForOpenCL library (described below). It is also possible, using ROSE, to modify the calling site so that the entire program can be transformed, but this functionality is outside the scope of this paper. Here we specifically examine transforming Fortran procedures to OpenCL kernels. Because OpenCL support is relatively new to ROSE, some generated code must be modified. For example, the `__global` attribute for kernel arguments was added by hand.

It is assumed that memory for all arrays reside on the device. The programmer must copy memory to and from the device. In addition, array size (neglecting ghost cell regions) must be multiples of the global OpenCL kernel size.

Array variables within a kernel procedure (specified by the `$OFP PURE, KERNEL` directive) must be declared as contiguous. A kernel procedure may not call other procedures except for limited intrinsic functions (primarily math), user-defined elemental functions, and the `region_cpy` and `region_ptr` functions. Future work will address non-contiguous arrays (such as those that result from strided access) by mapping array strides to gather/scatter-style memory accessors.

Array parameters to a kernel procedure must be declared as either intent(in) or intent(out); they cannot be intent(inout). A thread may read from an extended region about its local element (using the `region_cpy` function), but can only write to the single array element it owns. If a variable were intent(inout), a thread could update its array element before another thread had read from that element. This restriction requires double buffering techniques.

3. SHALLOW WATER MODEL

The numerical code used for this work is from a presentation at the NM Supercomputing Challenge [13]. The algorithm solves the standard 2D shallow water equations. This algorithm is typical of a wide range of modeling equations based on conservation laws such as compressible fluid dynamics (CFD), elastic material waves, acoustics, electromagnetic waves and even traffic flow [11]. For the shallow water problem there are three equations with one based on conservation of mass and the other two on conservation of momentum.

$$\begin{aligned} h_t + (hu)_x + (hv)_y &= 0 \quad \text{(mass)} \\ (hu)_t + (hu^2 + \tfrac{1}{2}gh^2)_x + (huv)_y &= 0 \quad (x\text{-momentum}) \\ (hv)_t + (huv)_x + (hv^2 + \tfrac{1}{2}gh^2)_y &= 0 \quad (y\text{-momentum}) \end{aligned}$$

where h = height of water column (mass), u = x velocity, v = y velocity, and g = gravity. The height h can be used for mass because of the simplification of a unit cell size and a uniform water density. Another simplifying assumption is that the water depth is small in comparison to length and width and so velocities in the z-direction can be ignored. A fixed time step is used for simplicity though it must be less than $dt \leqq dx/(\sqrt{gh} + |u|)$ to fulfill the CFL condition.

The numerical method is a two-step Lax-Wendroff scheme. The method has some numerical oscillations with sharp gradients but is adequate for simulating smooth shallow-water flows. In the following explanation, U is the conserved state variable at the center of the cell. This state variable, $U = (h, hu, hv)$ in the first term in the equations below. F is the flux quantity that crosses the boundary of the cell and is subtracted from one cell and added to the other. The remaining terms after the first term are the flux terms in the equations above with one term for the flux in the x-direction and the next term for the flux in the y-direction. The first step estimates the values a half-step advanced in time and space on each face, using loops on the faces.

$$\begin{aligned} U^{n+\frac{1}{2}}_{i+\frac{1}{2},j} &= (U^n_{i+1,j} + U^n_{i,j})/2 + \frac{\triangle t}{2\triangle x}\left(F^n_{i+1,j} - F^n_{i,j}\right) \\ U^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} &= (U^n_{i,j+1} + U^n_{i,j})/2 + \frac{\triangle t}{2\triangle y}\left(F^n_{i,j+1} - F^n_{i,j}\right) \end{aligned}$$

The second step uses the estimated values from step 1 to compute the values at the next time step in a dimensionally unsplit loop.

$$U^{n+1}_{i,j} = U^n_{i,j} - \frac{\triangle t}{\triangle x}(F^{n+\frac{1}{2}}_{i+\frac{1}{2},j} - F^{n+\frac{1}{2}}_{i-\frac{1}{2},j}) - \frac{\triangle t}{\triangle y}(F^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} - F^{n+\frac{1}{2}}_{i,j-\frac{1}{2}})$$

3.1 Fortran implementation

Selected portions of the data-parallel implementation of the shallow water model are now shown. This code serves as input to the ForOpenCL transformations described in the next section. The interface for the Fortran kernel procedure `wave_advance` is declared as:

```
subroutine wave_advance(dx,dy,dt,H,U,V,oH,oU,oV)
  !$OFP PURE, KERNEL   :: wave_advance
  real, intent(in)     :: dx,dy,dt
  real, dimension(:,:) :: H,U,V,oH,oU,oV
  contiguous  :: H,U,V,oH,oU,oV
  intent(in)  :: H,U,V
  intent(out) :: oH,oU,oV
  target      :: oH,oU,oV
end subroutine
```

where `dx, dy, dt` are differential quantities in space x, y and time t, `H, U`, and `V` are state variables for the height and x and y momentum respectively, and `oH, oU, oV` are corresponding output arrays used in the double buffering scheme. The *OFP* compiler-directive attributes `PURE` and `KERNEL` indicate that the procedure `wave_advance` is to be transformed as an OpenCL kernel and that it must be pure, other than for any pointers used to reference interior regions of the output arrays.

Temporary arrays are required for the quantities `Hx, Hy, Ux, Vx`, and `Vy`, that are defined on cell faces. Also, the pointer variables, `pH, pU`, and `pV`, are needed to access and update interior regions of the output arrays. As these pointers are assigned to the arrays `oH, oU, oV`, these output arrays must have the target attribute, as shown in the interface above. The temporary arrays and array pointers are declared as,

```
real, allocatable, dimension(:,:) :: Hx, Hy, Ux
real, allocatable, dimension(:,:) :: Uy, Vx, Vy
real, pointer,     dimension(:,:) :: pH, pU, pV
```

Halo variables for the interior and the cell faces are declared and defined as

```
integer, dimension(4) :: face_lt, face_rt, halo
integer, dimension(4) :: face_up, face_dn

halo    = [1,1,1,1]
face_lt = [0,1,1,1];  face_rt = [1,0,1,1]
face_dn = [1,1,0,1];  face_up = [1,1,1,0]
```

Note that the halo definitions for the four faces each have a 0 in the initialization. Thus the returned array copy will have a size that is larger than any interior region that uses the full halo `[1,1,1,1]`. This is because there is one more cell face quantity than there are cells in a given direction.

The first Lax-Wendroff step updates state variables on the cell faces. Assignment statements like the following,

```
Hx = 0.5*( region_cpy(H,face_lt) +   &
           region_cpy(H,face_rt) )   &
     + (0.5*dt/dx)                   &
     * (region_cpy(U,face_lt) - region_cpy(U,face_rt))
```

are used to calculate these quantities. This equation updates the array for the height in the x-direction. The second step then uses these face quantities to update the interior region, for example,

```
face_lt = [0,1,0,0];  face_rt = [1,0,0,0]
face_dn = [0,0,0,1];  face_up = [0,0,1,0]

pH = region_ptr(oH, halo)

pH = region_cpy(H, halo)
       + (dt/dx) * ( region_cpy(Ux, face_lt) -   &
                     region_cpy(Ux, face_rt) )   &
       + (dt/dy) * ( region_cpy(Vy, face_dn) -   &
                     region_cpy(Vy, face_up) )
```

Note that face halos have been redefined so that the array copy returned has the same size as the interior region.

These simple code segments show how the shallow water model is implemented in standard Fortran using the data-parallel programming model described above. The resulting code is simple, concise, and easy to understand. However it does *not* necessarily perform well when compiled for a traditional sequential system because of suboptimal use of temporary array variables, especially those produced by the function `region_cpy`. This is generally true of algorithms that use Fortran shift functions as well, as some Fortran compilers (e.g., gfortran) do not generate optimal code for shifts. We note (as shown below) that these temporary array copies are replaced by scalars in the transformed Fortran code so there are no performance penalties for using data-parallel statements as outlined. However, there is an increased memory cost due to the double buffering required by the kernel execution semantics.

4. SOURCE-TO-SOURCE TRANSFORMATIONS

This section provides an brief overview of the ForOpenCL transformations that take Fortran elemental and pure procedures as input and generate OpenCL code. Elemental functions are transformed to inline OpenCL functions and subroutines with the `PURE` and `KERNEL` compiler directive attributes are transformed to OpenCL kernels.

4.1 OpenCL

OpenCL [9] is an open-language standard for developing applications targeted for GPUs, as well as for multi-threaded applications targeted for multi-core CPUs. The kernels are run by calling a C runtime library from the OpenCL host (normally the CPU). Efforts to standardize a C++ runtime are underway and Fortran interfaces to the C runtime are distributed in the ForOpenCL library.

An important concept in OpenCL is that of a thread and a thread group. Thread groups are used to run an OpenCL kernel concurrently on several processor elements on the OpenCL device (often a GPU). Consider a data-parallel statement written in terms of an elemental function as discussed above. The act of running an OpenCL kernel can be thought of as having a particular thread assigned to each instance of the call to the elemental function as it is mapped across the arrays in the data-parallel statement. In practice, these threads are packaged into thread groups when they are run on the device hardware.

Device memory is separated hierarchically. A thread instance has access to its own thread memory (normally a

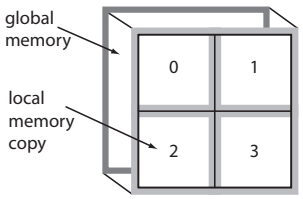


Figure 1: A schematic of global memory for an array and its copy stored in local memory for four thread groups.

set of registers), threads in a thread group to OpenCL local memory, and all thread groups have access to OpenCL global memory. When multiple members of a thread group access the same memory elements (for example in the use of the `region_cpy` function in the calculation of face variable quantities shown above), for performance reasons it is often best that global memory accessed and shared by a thread group be copied into local memory.

The *region* and *halo* constructs easily map onto the OpenCL memory hierarchy. A schematic of this mapping is shown in Figure 1 for a two-dimensional array with a 2x2 array of 4 thread groups. The memory for the array and its halo are stored in global memory on the device as shown in the background layer of the figure. The array copy in local memory is shown in the foreground divided into 4 *local* tiles that partition the array. Halo regions in global memory are shown in dark gray and halo regions in local memory are shown in light gray.

We point out that the hierarchical distribution of memory used on the OpenCL device shown in Figure 1 is similar to the distribution of memory across MPI nodes in an MPI application. In the case of MPI, the virtual global array is represented by the background layer (with its halo) and partitions of the global array are stored in the 4 MPI nodes shown in the foreground. Our current and future work on this effort includes source-to-source transformations to generate MPI code in addition to OpenCL in order to deal with clusters of nodes containing accelerators. This work is outside the scope of this paper.

Halo regions obtained via the `region_cpy` function (used with intent(in) arrays) are constrained semantically so that they can not be written to by an OpenCL kernel. The `region_cpy` function returns a copy of the region of the array stored in global memory and places it in local memory shared by threads in a thread group. Thus once memory for an array has been transferred into global device memory by the host (before the OpenCL kernel is run), memory is in a consistent state so that all kernel threads are free to read from global device memory. Because the local memory is a copy, it functions as a software cache for the local thread group. Thus the compiler must insert OpenCL barriers at proper locations in the code to insure that all threads have written to the local memory cache before any thread can start to read from the cache. On exit from a kernel, any local memory explicitly stored in register variables by the compiler (memory accessed via the `region_cpy` function) is copied back to global memory for all intent(out) arrays. Recall that a thread may only write to its own intent(out) array element, thus there are no race conditions when updating intent(out) arrays.

4.2 Transformation examples

This section outlines the OpenCL equivalent syntax for portions of the Fortran shallow-water code described in Section 3. The notation uses uppercase for arrays and lowercase for scalar quantities. Variables temporarily storing quantities for updated output arrays (declared as pointers in Fortran) are denoted by a `p` preceding the array name. For example, the Fortran statement `pH = region_ptr(oH, halo)` is transformed as a scalar variable declaration representing a single element in the output array `oH`.

4.2.1 Region function

While the Fortran version of the `region_cpy` function semantically returns an array copy, in OpenCL this function returns a scalar quantity based on the location of a thread in a thread group and the relationship of its location to the array copy transferred to local memory. Because we assume there is a thread for every element in the interior, the array index is just the thread index adjusted for the size of the halo. Thus `region_cpy` is just an inline OpenCL function and is provided by the ForOpenCL library.

4.2.2 Function and variable declarations

Fortran kernel procedures have direct correspondence with OpenCL equivalents. For example, the `wave_advance` interface declaration is transformed as

```
__kernel void
wave_advance(float dx, ..., __global float * H, ...);
```

The intent(in) arrays have local equivalents that are stored in local memory and are declared by, for example,

```
__local float H_local[LOCAL_SIZE];
```

These local arrays are declared with the appropriate size and are copied to local memory by the compiler with an inlined library function. The array temporaries defined on cell faces are declared similarly while interior pointer variables are simple scalars, e.g., `float pH`. Intent(in) array variables cannot be scalar objects because regions may be shifted and thus *shared* by threads within a thread group.

4.2.3 Array syntax

Array syntax transforms nearly directly to OpenCL code. For example, interior pointer variables are particularly straightforward as they are scalar quantities in OpenCL,

```
pH = region_cpy(H, halo)
       + (dt/dx) * ( region_cpy(Ux, face_lt) -
                     region_cpy(Ux, face_rt) )
       + (dt/dy) * ( region_cpy(Vy, face_dn) -
                     region_cpy(Vy, face_up) );
```

Array width	F90	GPU (16x8)	Speedup
16	0.025 ms	0.017 ms	1.5
32	0.086	0.02	4.3
64	0.20	0.02	10.0
128	0.76	0.036	21.1
256	3.02	0.092	32.8
512	12.1	0.32	37.8
1024	49.5	1.22	40.6
1280	77.7	1.89	41.1
2048	199.1	4.82	41.3
4096	794.7	19.29	41.2

Table 1: Performance measurements for the shallow-water code. All times reported in milliseconds.

Allocated variables are more complicated because they are arrays.

```
Hx[i] = 0.5 * (region(H_local, face_lt)+ ...);
```

where `i = LX + LY*(NLX+halo(0)+halo(1))` is a local index variable, `LX = get_local_id(0)` is the local thread id in the x dimension, `LY = get_local_id(1)` is the local thread id in the y dimension, `NLX = get_local_size(0)` is the size of the thread group in the x dimension, and the `get_local_id` and `get_local_size` functions are defined by the OpenCL language standard.

5. PERFORMANCE MEASUREMENTS

Performance measurements were made comparing the transformed code with different versions of the serial shallow-water code. The serial versions included two separate Fortran versions: one using data-parallel notation and the other using explicit looping constructs. We also compared with a hand-written OpenCL implementation that was optimized for local memory usage (no array temporaries). The accelerated measurements were made using an NVIDIA Tesla C2050 (Fermi) cGPU with 2.625 GB GDDR5 memory, and 448 processor cores. The serial measurements were made using an Intel Xeon X5650 hexacore CPU with 96 GB of RAM running at 2.67 GHz. The compilers were gfortran and gcc version 4.4.3 with an optimization level of -O3.

Several timing measurements were made by varying the size of the array state variables. The performance measurements are shown in Table 1. An average time was obtained by executing 100 iterations of the outer time-advance loop that called the OpenCL kernel. This tight loop kept the OpenCL kernel supplied with threads to take advantage of potential latency hiding by the NVIDIA GPU. Any serial code within this loop (not present in this study) would have reduced the measured values.

The transformed code achieved very good results. In all instances, the performance of the transformed code was within 25% of the hand-optimized OpenCL kernel. Most of the extra performance of the hand-optimized code can be attributed to the absence of array temporaries and to packing the three state variables `H, U`, and `V` into a single vector datatype.

While we did not have an OpenMP code for multicore comparisons, the transformed OpenCL code on the NVIDIA C2050 was up to 40 times faster than the best serial Fortran code executing on the host CPU.

6. CONCLUSIONS

The sheer complexity of programming for clusters of many or multi-core processors with tens of millions threads of execution makes the simplicity of the data-parallel model attractive. The increasing complexity of today's applications (especially in light of the increasing complexity of the hardware) and the need for portability across architectures make a higher-level and simpler programming model like data-parallel attractive.

The goal of this work has been to exploit source-to-source transformations that allow programmers to develop and maintain programs at a high-level of abstraction, without coding to a specific hardware architecture. Furthermore these transformations allow multiple hardware architectures to be targeted without changing the high-level source. It also removes the necessity for application programmers to understand details of the accelerator architecture or to know OpenCL.

7. ACKNOWLEDGMENTS

This work was supported in part by the Department of Energy Office of Science, Advanced Scientific Computing Research.

8. REFERENCES

[1] S. Benkner, G. Lonsdale, and H. Zima. The HPF+ Project: Supporting HPF for Advanced Industrial Applications. In *Proceedings of Euro-Par'99*, 1999.

[2] G. Blelloch and G. W. Sabot. Compiling collection oriented languages onto massively parallel processors. *Journal of Parallel and Distributed Computing*, 8(2), Feb 1990.

[3] R. G. Brickner, K. Holian, B. Thiagarajan, and S. L. Johnsson. Designing a stencil compiler for the connection machine model CM-5. Technical Report LA-UR-94-3152, LANL, 1994.

[4] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[5] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and L. Snyder. The High-Level Parallel Language ZPL Improves Productivity and Performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[6] A. D. Falkoff and K. E. Iverson. APL language summary. *SIGPLAN Notices*, 14(4), 1979.

[7] M. Feldman. Intel touts manycore coprocessor at supercomputing conference. *HPC-Wire*, 2011.

[8] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of ICFP 2010*, 2010.

[9] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[10] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[11] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics, 2002.

[12] J. Levesque. Fifty Years of Fortran. Panel Discussion, SuperComputing 2008, Reno, Nevada.

[13] R. W. Robey, R. Roberts, and C. Moler. Secrets of supercomputing: The conservation laws, supercomputing challenge kickoff. Research Report LA-UR-07-6793, LANL, 2007.

[14] The Portland Group. PGI Fortran & C Accelerator Programming Model. Technical report, March 2010.

[15] M. Wolfe. How We Should Program GPGPUs. *Linux Journal*, 2008.

Optimizing OpenCL Kernels for Iterative Statistical Applications on GPUs

Thilina Gunarathne
Indiana University,
Bloomington, IN 47405, USA
tgunarat@cs.indiana.edu

Bimalee Salpitikorala
Indiana University,
Bloomington, IN 47405, USA
ssalpiti@cs.indiana.edu

Arun Chauhan
Indiana University,
Bloomington, IN 47405, USA
achauhan@cs.indiana.edu

Geoffrey Fox
Indiana University,
Bloomington, IN 47405, USA
gcf@cs.indiana.edu

ABSTRACT

We present a study of three important kernels that occur frequently in iterative statistical applications: K-Means, Multi-Dimensional Scaling (MDS), and PageRank. We implemented each kernel using OpenCL and evaluated their performance on an NVIDIA Tesla GPGPU card. By examining the underlying algorithms and empirically measuring the performance of various components of the kernel we explored the optimization of these kernels by four main techniques: (1) caching invariant data in GPU memory across iterations, (2) selectively placing data in different memory levels, (3) rearranging data in memory, and (4) dividing the work between the GPU and the CPU. The optimizations resulted in performance improvements of up to 5X, compared to naïve OpenCL implementations. We believe that these categories of optimizations are also applicable to other similar kernels. Finally, we draw several lessons that would be useful in not only implementing other similar kernels with OpenCL, but also in devising code generation strategies in compilers that target GPGPUs through OpenCL.

1. INTRODUCTION

Iterative algorithms are at the core of the vast majority of scientific applications, which have traditionally been parallelized and optimized for large multi-processors, either based on shared memory or clusters of interconnected nodes. As GPUs have gained popularity for scientific applications, computational kernels used in those applications need to be performance-tuned for GPUs in order to utilize the hardware as effectively as possible.

Often, when iterative scientific applications are parallelized they are naturally expressed in a bulk synchronous parallel (BSP) style, where local computation steps alternate with collective communication steps [26]. An important class of such iterative applications are statistical applications that process large amounts of data. A crucial aspect of large data processing applications is that they can often be fruitfully run in large-scale distributed computing environments, such as clouds.

In this paper, we study three algorithms, which we refer to as *kernels*, that find use in such iterative statistical applications. The intended environment to run these applications is loosely-connected and distributed, which could be leveraged using a cloud computing framework, such as MapReduce. In this paper, we focus on characterizing and optimizing the kernel performance on a single GPU node. The three kernels are:

1. K-Means, which is a clustering algorithm used in many machine learning applications;
2. MDS, which is a set of statistical techniques to visualize higher dimensional data in three dimensions; and
3. PageRank, which is an iterative link analysis algorithm relying on sparse matrix-vector multiplication.

These kernels are characterized by high ratio of memory accesses to floating point operations, thus necessitating careful latency hiding and memory hierarchy optimizations to achieve high performance. We conducted our study in the context of OpenCL, which would let us extend our results across hardware platforms. We studied each kernel for its potential for optimization by:

1. Caching invariant data in GPU memory to be used across kernel invocations (i.e., algorithm iterations);
2. Utilizing OpenCL *local memory*, by software-controlled caching of selected data;
3. Reorganizing data in memory, to encourage hardware-driven memory access coalescing or to avoid bank conflicts; and
4. Dividing the computation between CPUs and GPUs, to establish a software pipeline across iterations.

We present detailed experimental evaluation for each kernel by varying different algorithmic parameters. Finally, we draw some lessons linking algorithm characteristics to the optimizations that are most likely to result in performance improvements. This has important implications not only for kernel developers, but also for compiler developers who wish to leverage GPUs within a higher level language by compiling it to OpenCL.

2. BACKGROUND

Boosted by the growing demand for gaming power, the traditional fixed function graphics pipeline of GPUs have evolved into a full-fledged programmable hardware chain [14].

In this paper we use NVIDIA Tesla C1060 GPGPU card for our experiments. Tesla C1060 consists of 240 processor cores and 4 GB global memory with 102 GB/sec peak memory bandwidth. It has a theoretical peak performance of 933 GFLOPS for single precision and 78 GFLOPS for double precision.

It is the general purpose relatively higher level programming interfaces, such as OpenCL, that have paved the way for leveraging GPUs for general purpose computing. OpenCL is a cross-platform, vendor-neutral, open programming standard that supports parallel programming in heterogeneous computational environments, including multi-core CPUs and GPUs [10]. It provides efficient parallel programming capabilities on both data parallel and task parallel architectures.

A *compute kernel* is the basic execution unit in OpenCL. Kernels are queued up for execution and OpenCL API provides a set of events to handle the queued up kernels. The data parallel execution of a kernel is defined by a multi-dimensional domain and each individual execution unit of the domain is referred to as a *work item*, which may be grouped together into several *work-groups*, executing in parallel. Work items in a group can communicate with each other and synchronize execution. The task parallel compute kernels are executed as single work items.

OpenCL defines a multi level memory model with four memory spaces: private, local, constant. and global as depicted in Figure 1. Private memory can only be used by single compute units, while global memory can be used by all the compute units on the device. Local memory (called *shared memory* in CUDA) is accessible in all the work items in a work group. Constant memory may be used by all the compute units to store read-only data.

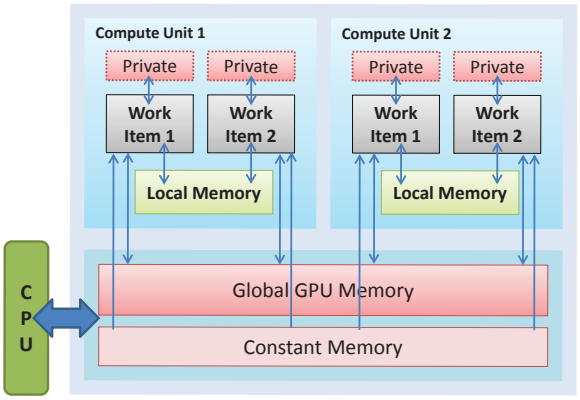


Figure 1: OpenCL memory hierarchy. In the current NVIDIA OpenCL implementation, private memory is physically located in global memory.

3. ITERATIVE STATISTICAL APPS

Many important scientific applications and algorithms can be implemented as iterative computation and communication steps, where computations inside an iteration are independent and are synchronized at the end of each iteration through reduce and communication steps. Often, each iteration is also amenable to parallelization. Many statistical applications fall in this category. Examples include clustering algorithms, data mining applications, machine learning algorithms, data visualization algorithms, and most of the expectation maximization algorithms. The growth of such iterative statistical applications, in importance and number, is driven partly by the need to process massive amounts of data, for which scientists rely on clustering, mining, and dimension-reduction to interpret the data. Emergence of computational fields, such as bioinformatics, and machine learning, have also contributed to an increased interest in this class of applications.

Advanced frameworks, such as Twister [9], can support optimized execution of iterative MapReduce applications, making them well-suited to support iterative applications in a large scale distributed environment, such as clouds. Within such frameworks, GPGPUs can be utilized for execution of single steps or single computational components. This gives the applications the best of both worlds by utilizing the GPGPU computing power and supporting large amounts of data. One goal of our current study is to evaluate the feasibility of GPGPUs for this class of applications and to determine the potential of combining GPGPU computing together with distributed cloud-computing frameworks. Some cloud-computing providers, such as Amazon EC2, are already moving to provide GPGPU resources for their users. Frameworks that combine GPGPU computing with the distributed cloud programming would be good candidates for implementing such environments.

Two main types of data can be identified in these statistical iterative applications, the loop-invariant input data and the loop-variant delta values. Most of the time, the loop-invariant input data, which remains unchanged across the iterations, are orders of magnitude larger than the loop-variant delta values. These loop-invariant data can be partitioned to process independently by ifferent worker threads. These loop-invariant data can be copied from CPU memory to GPU global memory at the beginning of the computation and can be reused from the GPU global memory across iterations, giving significant advantages in terms of the CPU to GPU data transfer cost. To this end, we restrict ourselves to scenarios where the loop-invariant computational data fit within the GPU memory, which are likely to be the common case in large-scale distributed execution environments consisting of a large number of GPU nodes. Loop-variant delta values typically capture the result of a single iteration and will be used in processing of the next iteration by all the threads, hence necessitating a broadcast type operation

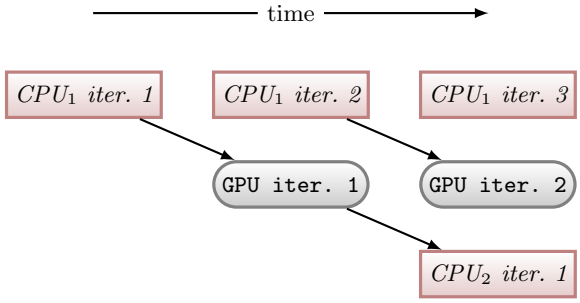


Figure 2: Software pipelining to leverage GPUs for loop-level parallelism.

of loop-variant delta values to all the worker threads at the beginning of each iteration. Currently we use global memory for this broadcast. Even though constant memory could potentially result in better performance, it is often too small to hold the loop-variant delta for the MDS and PageRank kernels we studied.

It is possible to use software pipelining for exploiting parallelism across iterations. Assuming that only one kernel can execute on the GPU at one time, Figure 2 shows a scheme for exploiting loop-level parallelism. This assumes that there are no dependencies across iterations. However, if the loop-carried dependence pattern is dynamic, i.e., it may or may not exist based on specific iterations or input data, then it is still possible to use a software pipelining approach to *speculatively* execute subsequent iterations concurrently and quashing the results if the dependencies are detected. Clearly, this sacrifices some parallel efficiency. Another scenario where such pipelining may be useful is when the loop-carried dependence is caused by a convergence test. In such a case, software pipelining would end up executing portions of iterations that were not going to be executed in the original program. However, that would have no impact on the converged result.

Note that if multiple kernels can be executed concurrently and efficiently on the GPU then the pipelining can be replicated to leverage that capability.

A characteristic feature of data processing iterative statistical applications is their high ratio of memory accesses to floating point operations, making them memory-bound. As a result, achieving high performance, measured in GFLOPS, is challenging. However, software-controlled memory hierarchy and the relatively high memory bandwidth of GPGPUs also offer an opportunity to optimize such applications. In the rest of the paper, we describe and study the optimization on GPUs of three representative kernels that are heavily used in iterative statistical applications. It should be noted that even though software pipelining served as a motivating factor in designing our algorithms, we did not use software pipelining for the kernels used in this study.

4. K-MEANS CLUSTERING

Clustering is the process of partitioning a given data set into disjoint clusters. Use of clustering and other data mining techniques to interpret very large data sets has become increasingly popular with petabytes of data becoming commonplace. Each partitioned cluster includes a set of data points that are *similar* by some clustering metric and differ from the set of data points in another cluster. K-Means clustering algorithm has been widely used in many scientific as well as industrial application areas due to its simplicity and the applicability to large data sets [20].

K-Means clustering algorithm works by defining k *centroids*, i.e., cluster means, one for each cluster, and associating the data points to the nearest centroid. It is often implemented using an iterative refinement technique, where each iteration performs two main steps:

1. In the cluster *assignment step*, each data point is assigned to the nearest centroid. The distance to the centroid is often calculated as Euclidean distance.
2. In the *update step*, new cluster centroids are calculated based on the data points assigned to the clusters in the previous step.

At the end of iteration n, the new centroids are compared with the centroids in iteration $n-1$. The algorithm iterates until the difference, called the *error*, falls below a predetermined threshold. Figure 3 shows an outline of our OpenCL implementation of the K-Means algorithm.

The number of floating-point operations, F, in OpenCL K-

```
__kernel KMeans(__global matrix,
  __global centroids,__global assignment,
  __local localPoints,__local localData){

 gid = get_global_id(0);
 lid = get_local_id(0);
 lz = get_local_size(0);

// Copying centroids to shared memory
 if (lid < centersHeight){
   for (int i=0; i < WIDTH ; i++){
    localPoints[(lid*WIDTH)+i] =
        centroids[(lid*WIDTH)+i];
   }
 }

// Copying data points to shared memory
 for (int i=0; i < WIDTH ; i++){
   localData[lid+(lz*i)] =
                matrix[(gid)+(i* height)];
 }
 for (int j = 0; j < centersHeight; j++){
   for (int i = 0; i < width; i++){
     distance = (localPoints[(j*width)+i]
                 - localData[lid +(lz*i)]);
     euDistance += distance * distance;
   }
   if (j == 0) {min = euDistance;}
   else if (euDistance < min) {
     min = euDistance; minCentroid = j;
   }
 }
 assignment[gid]=minCentroid;
}
```

Figure 3: Outline of K-Means in OpenCL.

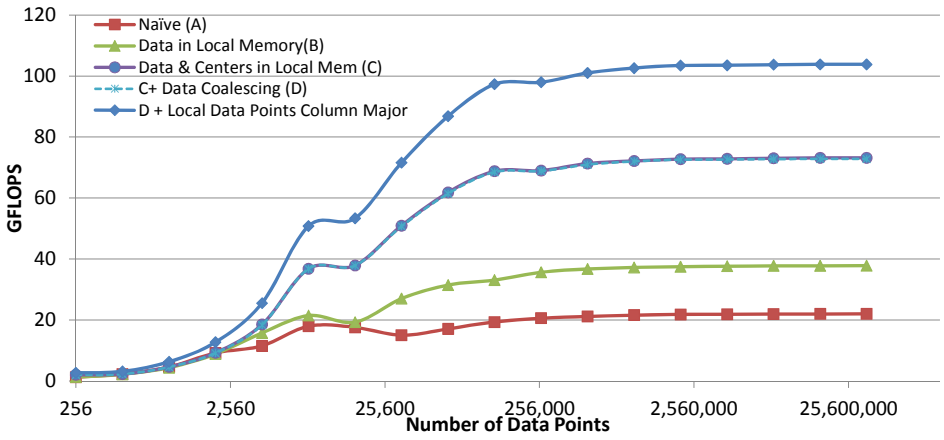


Figure 4: K-Means performance with the different optimizations steps, using 2D data points and 300 centroids.

Means per iteration per thread is given by $F = (3DM + M)$, resulting in a total of $F * N * I$ floating-point operations per calculation, where I is the number of iterations, N is the number of data points, M is the number of centers, and D is the dimensionality of the data points.

Figure 4 summarizes the performance of our K-Means implementation using OpenCL, showing successive improvements with optimizations. We describe these optimizations in detail in the remainder of this section.

4.1 Caching Invariant Data

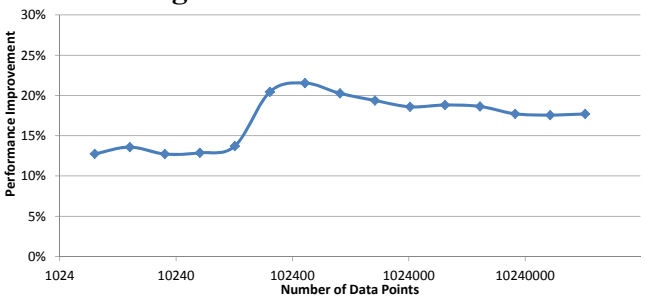


Figure 5: Performance improvement in K-Means due to caching of invariant data in GPU memory.

Transferring input data from CPU memory to GPU memory incurs major cost in performing data intensive statistical computations on GPUs. The speedup on GPU over CPU should be large enough to compensate for this initial data transfer cost. However, statistical iterative algorithms have loop-invariant data that could be reused across iterations. Figure 5 depicts the significant performance improvements gained by reusing of loop-invariant data in K-Means compared with no data reuse (copying the loop-invariant data from CPU to GPU in every iteration).

4.2 Leveraging Local Memory

In the naïve implementation, both the centroid values as well as the data points are accessed directly from the GPU global memory, resulting in a global memory read for each data and centroid data point access. With this approach, we were able to achieve performance in the range of 20 GFLOPs and speedups in the range of 13 compared to single core CPU[1].

The distance from a data point to each cluster centroid gets calculated in the assignment step of K-Means, resulting in reuse of the data point many times within a single thread. This observation motivated us to modify the kernel to copy the data points belonging to a local work group to the local memory, at the beginning of the computation. This resulted in approximately 75% performance increase over the naïve implementation, as the next line, marked "B", shows.

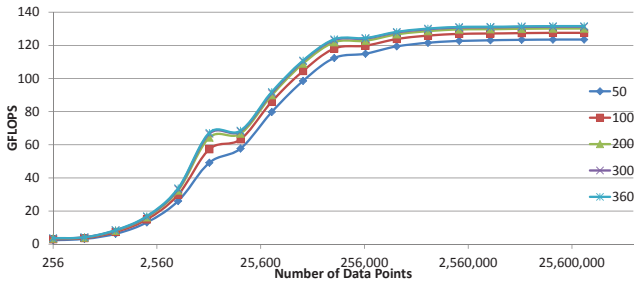
Each thread iterates through the centroids to calculate the distance to the data point assigned to that particular thread. This results in several accesses (equal to the local work group size) to each centroid per local work group. To avoid that, we copied the centroid point to the local memory before the computation. Caching of centroids values in local memory resulted in about 160% further performance increase, illustrated in the line marked "C" in Figure 4.

The performance curves changes at 8192 data point in Figure 4. We believe that this is due to the GPU getting saturated with threads at 8192 data points and above, since we spawn one thread for each data point. For data sizes smaller than 8192, the GPU kernel computation took a constant amount of time, indicating that GPU might have been underutilized for smaller data sizes. Finally, the flattening of the curve for large data sizes is likely because of reaching memory bandwidth limits.

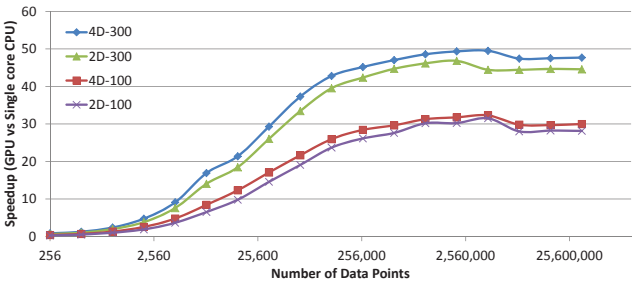
4.3 Optimizing Memory Access

As the next step, we stored the multi-dimensional data points in column-major format in global memory to take advantage of the hardware coalescing of memory accesses. However, this did not result in any measurable performance improvement as the completely overlapped lines "C" and "D" show,

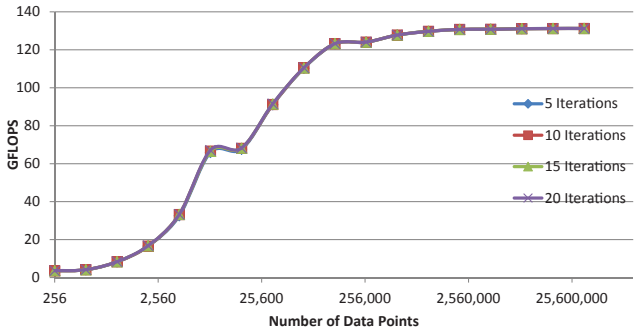
[1]We use a 3 GHz Intel Core 2 Duo Xeon processor, with 4 MB L2 cache and 8 GB RAM, in all our experiments.



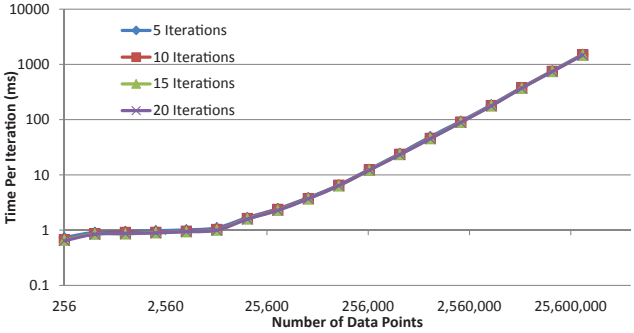
(a) K-Means: varying number of centers, using 4D data points.



(b) K-Means: varying number of dimensions.



(c) K-Means: varying number of iterations.



(d) K-Means (per iteration): varying number of iterations.

Figure 6: K-Means with varying algorithmic parameters.

in Figure 4.

However, storing the data points in local memory in column-major format resulted in about 140% performance improvement, relative to the naïve implementation, represented by the line marked "D + shared data points ..." in Figure 4. We believe that this is due to reduced bank conflicts when different threads in a local work group access local memory concurrently. Performing the same transformation for centroids in local memory did not result in any significant change to the performance (not shown in the figure). We believe this is because all the threads in a local work group access the same centroid point at a given step of the computation, resulting in a bank-conflict free broadcast from the local memory. All experiments for these results were obtained on a two-dimensional data set with 300 centroids.

Next, we characterized our most optimized K-Means algorithm by varying the different algorithmic parameters. Figure 6(a) presents the performance variation with different number of centroids, as the number of data points increases. Figure 6(b) shows the performance variation with 2D and 4D data sets, each plotted for 100 and 300 centroids. The measurements indicate that K-Means is able to achieve higher performance with higher dimensional data. Finally, Figures 6(c) and 6(d) show that there is no measurable change in performance with the number of iterations.

4.4 Sharing Work between CPU and GPU

In the OpenCL K-Means implementation, we follow a hybrid approach where cluster assignment step is performed in the GPU and the centroid update step is performed in the CPU. A single kernel thread calculates the centroid assignment for one data point. These assignments are then transfered back to the CPU to calculate the new centroid values. While some recent efforts have found that performing all the computation on the GPU can be beneficial, especially, when data sets are large [8], that approach forgoes the opportunity to make use of the powerful CPU cores that might also be available in a distributed environment. Performing partial computation on the CPU allows our approach to implement software pipelining within iteration by interleaving the work partitions and across several iterations through speculation.

4.5 Overhead Estimation

We used a simple performance model in order to isolate the overheads caused by data communication and kernel scheduling. Suppose that c_s is the time to perform K-Means computation and o_s is the total overheads (including data transfer to and from GPU and thread scheduling), for s data points. Then, the total running time of the algorithm, T_s is given by:

$$T_s = c_s + o_s \quad (1)$$

Suppose that we double the computation that each kernel thread performs. Since the overheads remain more or less unchanged, the total running time, T'_s, with double the computation is given by:

$$T'_s = 2{\cdot}c_s + o_s \quad (2)$$

By empirically measuring T_s and T'_s and using Equations 1 and 2, we can estimate the overheads. Figure 7 shows T'_s

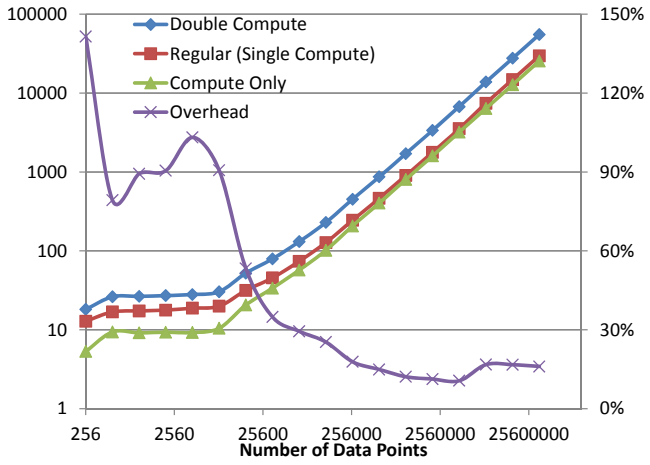


Figure 7: Overheads in OpenCL KMeans.

("double compute"), T_s ("regular"), c ("compute only") and o ("overhead"). The running times are in seconds (left vertical axis) and overhead is plotted as a percentage of the compute time, c (right vertical axis). Clearly, for small data sets the overheads are prohibitively high. This indicates that, in general, a viable strategy to get the best performance would be to offload the computation on the GPU only when data sets are sufficiently large. Empirically measured parameters can guide the decision process at run time.

5. MDS

The objective of multi-dimensional scaling (MDS) is to map a data set in high-dimensional space to a user-defined lower dimensional space with respect to pairwise proximity of the data points [16, 5]. Dimensional scaling is used mainly in visualization of high-dimensional data by mapping them to two or three dimensional space. MDS has been used to visualize data in diverse domains, including, but not limited to, bio-informatics, geology, information sciences, and marketing.

One of the popular algorithms to perform MDS is Scaling by MAjorizing a COmplicated Function (SMACOF) [7]. SMACOF is an iterative majorization algorithm to solve MDS problem with STRESS criterion, which is similar to expectation-maximization. In this paper, we implement the parallel SMACOF algorithm described by Bae et al [1].

The input for MDS is an $N \times N$ matrix of pairwise proximity values, where N is the number of data points in the high-dimensional space. The resultant lower dimensional mapping in D dimensions, called the `X` values, is an $N \times D$ matrix. For the purposes of this paper, we performed an unweighted mapping resulting in two main steps in the algorithm: (a) calculating new `X` values, and (b) calculating the stress of the new `X` values. There needs to be a global barrier between the two steps as stress value calculation requires all of the new `X` values. However the reduction step for `X` values in MDS is much simpler than in K-Means. Since each data point, k, independently produces the value `X[k]`, the reduction step reduces to simple aggregation in memory. Figure 8 outlines our OpenCL implementation of MDS.

```
__kernel MDS(__global float* data,
  __global float* x,__global float* newX){

  gid = get_global_id(0);

  for (int j = 0; j < WIDTH ; j++)
  {
    distance  = dist(x[gid][], x[j][]);
    bofZ = k * (data[gid][j] /distance);
    rowSum += bofZ;
    newX[gid][] += bofz * x[j][];
  }

  newX[gid][] += k * rowSum * x[gid][];
  newX[gid][] = newX[gid][]/WIDTH;

  barrier(CLK_GLOBAL_MEM_FENCE);

  __local float sigma[];
  for (int j = 0; j < WIDTH; j++)
  {
    distance = dist(newX[gid][], newX[j][])
    sigma[lid] += (data[gid][j]-distance)^2;
  }

  stress = hierachicalReduction(sigma[]);
}
```

Figure 8: Outline of MDS in OpenCL.

The number of floating pointer operations, F, per iteration per thread is given by $F = (8DN+7N+3D+1)$, resulting in a total of $F \times N \times I$ floating point operations per calculation, where I is the number of iterations, N is the number of data points, and D is the dimensionality of the lower dimensional space.

5.1 Caching Invariant Data

Similar to K-Means, MDS also has loop-invariant data that can fit in available global memory and that can be reused across iterations. Figure 9 summarizes the benefits of doing that for MDS.

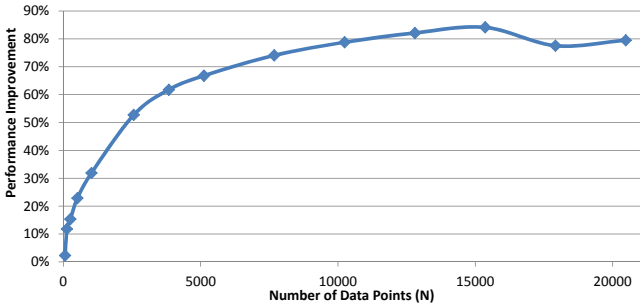
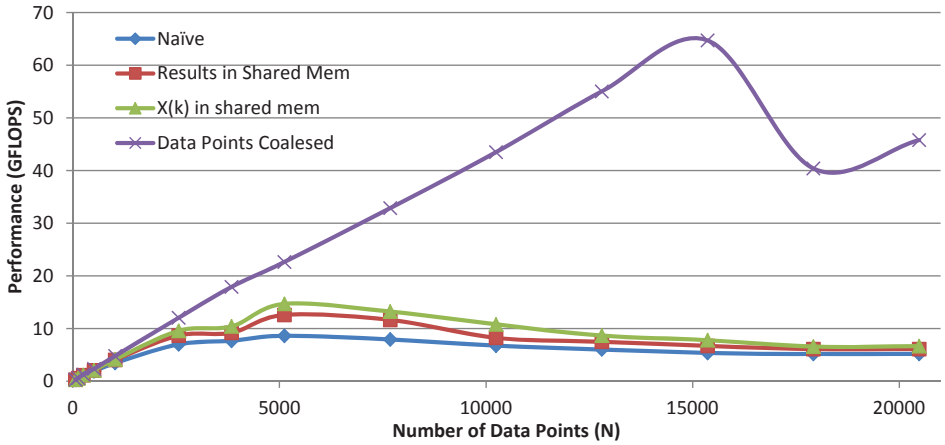


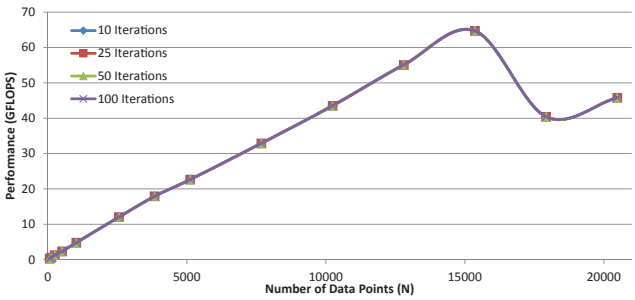
Figure 9: Performance improvement in MDS due to caching of invariant data in GPU memory.

5.2 Leveraging Local Memory

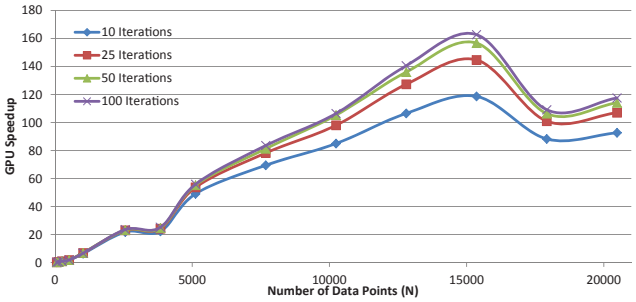
In a naïve implementation all the data points, `X` values, and result (new `X` values) are stored in global memory. SMACOF MDS algorithm uses a significant number of temporary runtime matrices for intermediate data storage. We restructured the algorithm to eliminate the larger temporary run time matrices, as they proved to be very costly in terms of



(a) MDS performance with the different optimizations steps.



(b) MDS: varying number of iterations.



(c) MDS per iteration: varying number of iterations.

Figure 10: MDS with varying algorithmic parameters.

space as well as performance. The kernel was redesigned to process a single row at a time.

After eliminating the run time data structures, `X'[k][]` matrix points were used in several locations of the algorithm to store intermediate results, which were stored in local memory and copied to global memory only at barrier synchronization. As an added advantage, we were able to reuse the intermediate values in local memory when calculating the stress values. This resulted in a significant performance improvement (up to about 45%) for intermediate size inputs as depicted in "Results in Shared Mem" curve of Figure 10(a).

`X[k]` values for each thread `k` were copied to local memory before the computation. `X` values belonging to the row that is being processed by the thread gets accessed many more times compared to the other `X` values. Hence, copying these `X` values to local memory turns out to be worthwhile. "X(k) in shared mem" curve of Figure 10(a) quantifies the gains.

5.3 Optimizing Memory Access

All data points belonging to the data row that a thread is processing are iterated through twice inside the kernel. We encourage hardware coalescing of these accesses by storing the data in global memory in column-major format, which causes contiguous memory access from threads inside a local work group. Figure 10(a) shows that data placement to encourage hardware coalescing results in a significant performance improvement.

We also experimented with storing the `X` values in column-major format, but it resulted in a slight performance degradation. The access pattern for the `X` values is different from that for the data points. All the threads in a local work group access the same `X` value at a given step. As we noted in Section 4.3, we observe a similar behavior with the K-Means clustering algorithm.

Performance improvements resulting from each of the above optimizations are summarized in Figure 10(a). Unfortunately, we do not yet understand why the performance drops suddenly after a certain large number of data points (peaks at 900 MB data size and drops at 1225 MB data size) and then begins to improve again. Possible explanations could include increased data bus contention, or memory bank conflicts. However, we would need more investigation to determine the exact cause. Figures 10(b) and 10(c) show performance numbers with varying number of iterations, which show similar trends.

5.4 Sharing Work between CPU and GPU

In the case of MDS, there is not a good case for dividing the work between CPU and GPU. In our experiments, the entire computation was done on the GPU. On the other hand, as the measured overheads show below, certain problem sizes might be better done on the CPU.

5.5 Overhead Estimation

Following the model that was used for K-Means in Section 4.5, we performed similar experiments for estimating kernel scheduling and data transfer overheads in MDS. Figure 11 shows the results. As in K-Means, we note that the overheads change with the input data size. In the case of MDS, however, there are two useful cutoffs, one for small data sizes and another for large data sizes—on either ends overheads become high and the computation might achieve higher performance on the CPU if the data have to be transferred from the CPU memory, which is what we have assumed in the overhead computations.

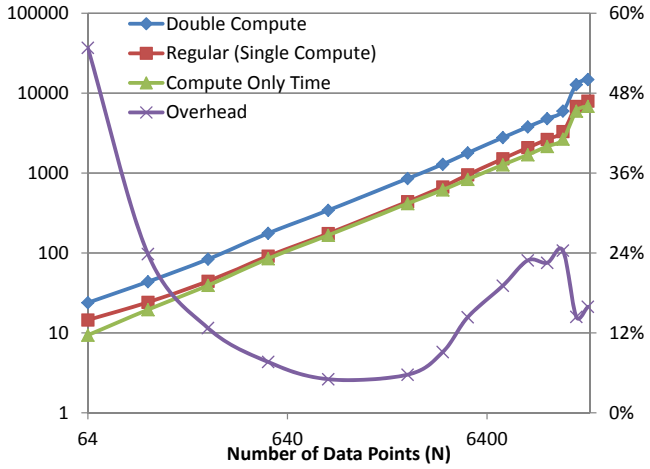


Figure 11: Overheads in OpenCL MDS.

6. PAGERANK

PageRank algorithm, developed by Page and Brin [6], analyzes linkage information of a set of linked documents to measure the relative importance of each document whithin the set. PageRank of a certain document depends on the number and the PageRank of other documents linked to it.

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \quad (3)$$

```
__kernel PageRankCSR(__global float* pointers,
  __global float* indices,__global float* x,
  __global float* newX){

  gid = get_global_id(0);

  start = pointers[(gid)];
  end = pointers[(gid)+1];

  for (int i=start; i < end ; i++)
  {
    newRank += ranks[indices[i]];
  }

  newRank = ((1-d)/numPages) + (d * newRank);
  // To avoid storing 1/L(pj) in the matrix.
  newRanks[gid] = newRank/numPages;
}
```

Figure 12: Outline of PageRank (CSR) in OpenCL.

Equation 3 defines PageRank, where $\{p_1, \ldots, p_N\}$ is the set of documents, $M(p_i)$ is the set of documents that link to p_i, $L(p_j)$ is the number of outbound links on p_j, and N is the total number of pages. PageRank calculation can be performed using an iterative power method, resulting in the multiplication of a sparse matrix and a vector. The linkage graph for the web is very sparse and follows a power law distribution [2], presenting unique implementation challenges for PageRank.

For our OpenCL PageRank implementation we used a modified compressed sparse row (CSR) format and modified ELLPACK format [4] to store the matrix representing the link graph. Typically the sparse matrix used for PageRank stores $1/L(p_j)$ in an additional *data* array. We eliminated the data array by storing the intermediate page rank values as $PR(p_j)/L(p_j)$, significantly reducing memory usage and accesses. We made a similar modification to ELLPACK format. We preprocessed and used the Stanford web data set from the Stanford Large Network Dataset [25] for our experiments. Our implementation is outlined in Figure 12.

6.1 Leveraging Local Memory

We were not able to utilize local memory to store all the data in the GPU kernel due to the variable sizes of matrix rows and the large size of the PageRank vector. However, we used local memory for data points in the ELLPACK kernel.

6.2 Optimizing Memory Access

Due to the irregular memory access pattern arising out of indirect array accesses, sparse matrix vector computation is not amenable to memory access optimizations. However, the index array, especially in the ELLPACK format, is stored in appropriate order to enable contiguous memory accesses.

6.3 Sharing Work between CPU and GPU

Due to the power law distribution of non-zero elements, a small number of rows contains a large number of elements, but a large number of rows are very sparse. In a preprocessing step, the rows are partitioned into two or more sets of those containing a small number of elements and the remainder containing higher number of elements. The more dense rows could be computed either on the CPU or the GPU using the CSR format directly. The rows with smaller number of non-zero elements are reformatted into the ELLPACK format and computed on the GPU. We evaluated several partitioning alternatives, shown in Figure 13.

The leftmost bars represent the running times on CPU. The next three bars represents computing all rows with greater than or equal to k elements on the CPU, where k is 4, 7, and 16, respectively. The rows with fewer than k elements are transformed into ELLPACK format and computed on the GPU. Moreover, when $k = 7$, two distinct GPU kernels are used, one for computing rows with up to 3 elements and another for computing rows with 4 to 7 elements. Similarly, for $k = 16$, an additional third kernel is used to process rows with 8 to 15 elements. Splitting the kernels not only improves the GPU occupancy, but also allows those kernels to be executed concurrently.

In Figure 13 we do not include the overheads of the linear time preprocessing step and of host-device data transfers,

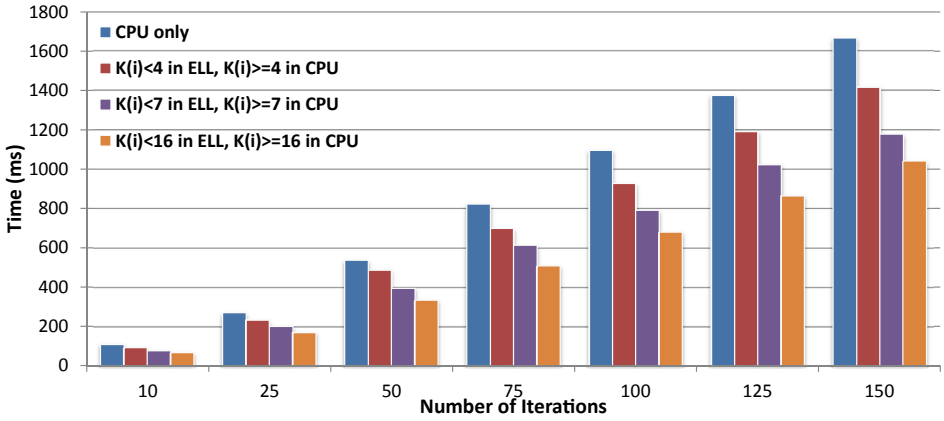


Figure 13: Potential implementations of PageRank.

both of which are relatively easy to estimate. However, we also do not assume any parallelism between the multiple kernels processing the rows in ELLPACK format. Our main observation from these experiments is that sharing work between CPU and GPU for sparse matrix-vector multiplication is a fruitful strategy. Moreover, unlike previous attempts recommending hybrid matrix representation that used a single kernel for the part of the matrix in ELLPACK format [4], our experiments indicate that it is beneficial to use multiple kernels to handle rows with different numbers of non-zero elements. The problem of deciding the exact partitioning and the exact number of kernels is outside the scope of this paper and we leave that as part of future work.

Instead of computing the matrix partition with denser rows on the CPU, it could also be computed on the GPU. We also implemented a sparse matrix-vector product algorithm using CSR representation on the GPU (not shown in the figure). Our experiments indicate that GPU can take an order of magnitude more time for that computation than CPU, underlining the role of CPU for certain algorithm classes.

7. LESSONS

In this study we set out to determine if we could characterize some core data processing statistical kernels for commonly used optimization techniques on GPUs. We focused on three widely used kernels and four important optimizations. We chose to use OpenCL, since there are fewer experimental studies on OpenCL, compared to CUDA, and the multi-platform availability of OpenCL would allow us to extend our research to other diverse hardware platforms. Our findings can be summarized as follows:

1. Since parts of the algorithms tend to employ sparse data structures or irregular memory accesses it is useful to carry out portions of computation on the CPU.

2. In the context of clusters of GPUs, inter-node communication needs to go through CPU memory (as of the writing of this paper in mid-2011). This makes computing on the CPUs a compelling alternative on data received from remote nodes, when the CPU-memory to device-memory data transfer times would more than offset any gains to be had running the algorithms on the GPUs.

3. Whenever possible, caching invariant data on GPU for use across kernel invocations significantly impacts performance.

4. While carefully optimizing the algorithms using specialized memory is important, as past studies have found, iterative statistical kernels cause complex trade-offs to arise due to irregular data access patterns (e.g., in use of texture memory) and size of invariant data (e.g., in use of constant memory).

5. Encoding algorithms directly in OpenCL turns out to be error-prone and difficult to debug. We believe that OpenCL might be better suited as a compilation target than a user programming environment.

In the rest of this section we elaborate on these findings.

Sharing work between CPU and GPU. One major issue in sharing work between CPU and GPU is the host-device data transfers. Clearly, this has to be balanced against the improved parallelism across GPUs and multi-core CPUs. Moreover, within the context of our study, there is also the issue of how data across nodes get transferred. If the data must move through CPU memory then in certain cases it might be beneficial to perform the computation on the CPU. Through our simple performance model and the overhead graphs the trade-offs are apparent. These graphs could also help in determining the cutoffs where offloading computation on the GPU is worthwhile. Finally, in iterative algorithms, where kernels are invoked repeatedly, offloading part of the computation on the GPUs can also enable software pipelining between CPU and GPU interleaving different work partitions.

Another factor in determining the division of work is the complexity of control flow. For instance, a reduction operation in K-Means, or a sparse matrix-vector multiply with relatively high density of non-zero values that might involve a reduction operation, may be better suited for computing

on the CPU. This would be especially attractive if there is sufficient other work to overlap with GPU computations.

Finally, the differences in precision between CPU and GPU can sometimes cause an iterative algorithm to require different number of iterations on the two. A decision strategy for scheduling an iterative algorithm between CPU and GPU may also need to account for these differences.

Unlike other optimizations, the value of this one is determined largely by the nature of input data. As a result, a dynamic mechanism to schedule computation just-in-time based on the category of input could be a more useful strategy than a static one.

GPU caching of loop-invariant data. There turns out to be a significant amount of data that are invariant and used across multiple kernel calls. Such data can be cached in GPU memory to avoid repeated transfers from the CPU memory in each iteration. However, in order to harness this benefit, the loop-invariant data should fit in the GPU global memory and should be retained throughout the computation. When the size of loop-invariant data is larger than the available GPU global memory, it is more advantageous to distribute the work across compute nodes rather than swapping the data in and out of the GPU memory.

Leveraging Local Memory. It is not surprising that making use of faster local memory turns out to be one of the most important optimizations within OpenCL kernels. In many cases, decision about which data to keep in local memory is straightforward based on reuse pattern and data size. For example, in K-Means and MDS it is not possible to keep the entire data set in local memory, since it is too big. However, the centroids in K-Means and intermediate values in MDS can be fruitfully stored there. Unfortunately, in some cases, such as portions of MDS, leveraging local memory requires making algorithmic changes in the code, which could be a challenge for automatic translators.

Leveraging Texture Memory. Texture memory provides a way to improved memory access of read-only data that has regular access pattern in a two-dimensional mapping of threads when the threads access contiguous chunks of a two-dimensional array. In the kernels we studied we found that the best performance was achieved when threads were mapped in one dimension, even when the array was two-dimensional. Each thread operated on an entire row of the array. As a result, our implementation was not conducive to utilizing texture memory.

Leveraging Constant Memory. As the name suggests, constant memory is useful for keeping the data that is invariant through the lifetime of a kernel. Unfortunately, the size of the constant memory was too small to keep the loop-invariant data, which do not change across kernel calls, for the kernels that we studied. However, since the data are required to be invariant only through one invocation of the kernel, it is possible to use constant memory to store data that might change across kernel calls as long as there is no change within one call of the kernel. The potential benefit comes when such data exhibit temporal locality, since the GPU has a hardware cache to store values read from constant memory so that hits in the cache are served much faster than misses. This gives us the possiblity to use constant memory for the broadcasting of loop-variant data, which are relatively small and do not change within a single iteration. Still the loop-variant data for larger MDS and PageRank test cases were larger than the constant memory size.

Optimizing Data Layout. Laying out data in memory is a known useful technique on CPUs. On GPUs, we observed mixed results. While data layout in local memory turned out to be useful for K-Means and not for MDS, layout in global memory had significant impact on MDS and no observable impact on K-Means. This behavior is likely a result of different memory access patterns. In general, contiguous global memory accesses encourage hardware coalescing, whereas on local memory bank conflicts play a more critical role. Thus, the two levels of memories require different layout management strategies. However, as long as the memory access patterns are known the benefits are predictable, thus making this optimization amenable to automatic translation.

OpenCL experience. OpenCL provides a flexible programing environment and supports simple synchronization primitives, which helps in writing substantial kernels. However, details such as the absence of debugging support and lack of dynamic memory allocation still make it a challenge writing code in OpenCL. One possible way to make OpenCL-based GPU computing accessible to more users is to develop compilers for higher level languages that target OpenCL. Insights gained through targeted application studies, such as this, could be a useful input to such compiler developers.

8. RELATED WORK

Emergence of accessible programming interfaces and industry standard languages has tremendously increased the interest in using GPUs for general purpose computing. CUDA, by NVIDIA, has been the most popular framework for this purpose [21]. In addition to directly studying application implementations in CUDA [11, 23, 30], there have been recent research projects exploring CUDA in hybrid CUDA/MPI environment [22], and using CUDA as a target in automatic translation [18, 17, 3].

There have been several past attempts at implementing the K-Means clustering algorithm on GPUs, mostly using CUDA or OpenGL [24, 12, 27, 19, 15]. Recently, Dhanasekaran et al. have used OpenCL to impelement the K-Means algorithm [8]. In contrast to the approach of Dhanasekaran et al., who implemented the reduction step on GPUs in order to handle very large data sets, we chose to mirror the earlier efforts with CUDA and perform the reduction step on the CPU. Even though that involves transferring the reduction data to CPU, we found that the amount of data that needed to be transferred was relatively small. In optimizing K-Means, we used the device shared memory to store the map data. As a result, when dealing with very large data

sets, which motivated Dhanasekaran et al.'s research, our optimized kernel would run out of shared memory before the reduction data becomes too large to become a bottleneck. Further research is needed to determine the trade-offs of giving up the optimization of device shared-memory and performing the reduction on the GPU.

We implemented the MDS kernel based on an SMACOF implementation by Bae et al. [1]. Glimmer is another multilevel MDS implementation [13]. While Glimmer implements multilevel MDS using OpenGL Shading Language (GLSL) for large data sets, Bae used an interpolated approach for large data sizes, which has been found to be useful in certain contexts. This allowed us to experiment with optimizing the algorithm for realistic contexts, without worrying about dealing with data sets that do not fit in memory. Our MDS implementation uses the SMACOF iterative majorization algorithm. SMACOF is expected to give better quality results than Glimmer, even though Glimmer can process much larger data sets than SMACOF [13]. Since our study is in the context of GPU clusters, with potentially vast amounts of distributed memory, we traded off in favor of a more accurate algorithm.

The computationally intensive part of PageRank is sparse matrix-vector multiplication. We followed the guidelines from an NVIDIA study for implementing the sparse matrix-vector multiplication [4]. The sparse matrix in PageRank algorithm usually results from graphs following power law. Recent efforts to optimize PageRank include using a low-level API to optimize sparse matrix-vector product by using the power law characteristics of the sparse matrix [28]. More recently, Yang et al. leveraged this property to auto-tune sparse matrix-vector multiplication on GPUs [29]. They built an analytical model of CUDA kernels and estimated parameters, such as tile size, for optimal execution.

9. CONCLUSION AND FUTURE WORK

We have presented an experimental evaluation of three important kernels used in iterative statistical applications for large scale data processing, using OpenCL. We evaluated three optimization techniques for each, based on leveraging fast local memory, laying out data for faster memory access, and dividing the work between CPU and GPU. We conclude that leveraging local memory is critical to performance in almost all the cases. Data layout is important in certain cases, but when it is, it has significant impact. In contrast to other optimizations, sharing work between CPU and GPU may be input data dependent, as in the case of K-Means, which points to the importance of dynamic just-in-time scheduling decisions.

Our planned future work includes extending the kernels to a distributed environment, which is the context that has motivated our study. Other possible directions include comparing the OpenCL performance with CUDA, studying more kernels from, possibly, other domains, and exploring more aggressive CPU/GPU sharing on more recent hardware that has improved memory bandwidth.

10. ACKNOWLEDGEMENTS

Thilina was supported by National Institutes of Health grant 5 RC2 HG005806-02. We thank Sueng-Hee Bae, BingJing Zang and Li Hui for the algorithmic insights they provided for the applications discussed in this paper. We thank our anonymous reviewers for providing constructive comments and suggestions to improve the paper.

11. REFERENCES

[1] S.-H. Bae, J. Y. Choi, J. Qiu, and G. C. Fox. Dimension reduction and visualization of large high-dimensional data via interpolation. In *Proceedings of the 19th ACM International Symposium on HIgh Performance Distributed Computing (HPDC)*, pages 203–214, 2010.

[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, Oct. 1999.

[3] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *In Proceedings of the 19th International Conference on Compiler Construction (CC)*, pages 244–263, 2010.

[4] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[5] I. Borg and P. J. F. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Statistics. Springer, second edition, 2005.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, volume 30 of *Computer Networks and ISDN Systems*, pages 107–117, Apr. 1998.

[7] J. de Leeuw. Convergence of the majorization method for multidimensional scaling. *Journal of Classification*, 5(2):163–180, 1988.

[8] B. Dhanasekaran and N. Rubin. A new method for GPU based irregular reductions and its application to k-means clustering. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4)*, 2011.

[9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance and Distributed Computing (HPDC)*, pages 810–818, 2010.

[10] K. Group. OpenCL: The open standard for parallel programming of heterogeneous systems. On the web. `http://www.khronos.org/opencl/`.

[11] T. R. Halfhill. Parallel processing with CUDA. *Microprocessor Report*, Jan. 2008.

[12] B. Hong-tao, H. Li-li, O. Dan-tong, L. Zhan-shan, and L. He. K-Means on commodity GPUs with CUDA. In *Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering*, 2009.

[13] S. Ingram, T. Munzner, and M. Olano. Glimmer: Multilevel MDS on GPU. *IEEE Transactions on Visualization and Computer Graphics*, 15(2):249–261, 2009.

[14] M. Johansson and O. Winter. General purpose computing on graphics processing units using OpenCL. Masters thesis, Chalmers University of Technology, Department of Computer Science and Engineering Göteborg, Sweden, June 2010.

[15] K. J. Kohlhoff, M. H. Sosnick, W. T. Hsu, V. S.

Pande, and R. B. Alteman. CAMPAIGN: An open-source library of GPU-accelerated data clustering algorithms. *Bioinformatics*, 2011.

[16] J. B. Kruskal and M. Wish. *Multidimensional Scaling*. Sage Publications Inc., 1978.

[17] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010)*, 2010.

[18] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[19] Y. Li, K. Zhao, X. Chu, and J. Liu. Speeding up K-Means algorithm by GPUs. In *Proceedings of the 2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*, pages 115–122, 2010.

[20] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, Mar. 1982.

[21] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide*, version 1.1 edition, Nov. 2007.

[22] S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. Mudalige. Performance analysis of a hybrid MPI/CUDA implementation of the NAS-LU benchmark. In *Proceedings of the First International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS)*, 2010.

[23] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 73–82, 2008.

[24] S. A. A. Shalom, M. Dash, and M. Tue. Efficient K-Means clustering using accelerated graphics processors. In *Proceedings of the 10th International Conference on Data Warehousing and Knowledge Discovery (DaWak)*, volume 5182 of *Lecture Notes in Computer Science*, pages 166–175, 2008.

[25] SNAP. Stanford network analysis project. On the web. http://snap.stanford.edu/.

[26] L. G. Valiant. Bulk-synchronous parallel computers. In M. Reeve, editor, *Parallel Processing and Artificial Intelligence*, pages 15–22. John Wiley & Sons, 1989.

[27] R. Wu, B. Zhang, and M. Hsu. GPU-accelerated large scale analytics. Technical Report HPL-2009-38, Hewlett Packard Lts, 2009.

[28] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu. Efficient PageRank and SpMV computation on AMD GPUs. In *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*, pages 81–89, 2010.

[29] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining. *Proceedings of the VLDB Endowment*, 4(4):231–242, 2011.

[30] H. Zhou, K. Lange, and M. A. Suchard. Graphics processing units and high-dimensional optimization. *Statistical Science*, 25(3):311–324, 2010.

A scalable hybrid algorithm based on domain decomposition and algebraic multigrid for solving partial differential equations on a cluster of CPU/GPUs

Li Luo
Shenzhen Inst. of Adv. Tech.
Chinese Academy of Sciences
Shenzhen 518055, P. R. China
li.luo@siat.ac.cn

Chao Yang
Inst. of Software
Chinese Academy of Sciences
Beijing 100190, P. R. China
yangchao@iscas.ac.cn

Yubo Zhao
Shenzhen Inst. of Adv. Tech.
Chinese Academy of Sciences
Shenzhen 518055, P. R. China
yb.zhao@siat.ac.cn

Xiao-Chuan Cai
Dept. of Computer Science
Univ. of Colorado at Boulder
Boulder, CO 80309, USA
cai@cs.colorado.edu

ABSTRACT

Several of the top ranked supercomputers are based on the hybrid architecture consisting of a large number of CPUs and GPUs. Very high performance has been obtained for problems with special structures, such as FFT-based image processing or N-body based particle calculations. However, for the class of problems described by partial differential equations discretized by finite difference (or other mesh based methods such as finite element) methods, obtaining even reasonably good performance on a CPU/GPU cluster is challenging. In this paper, we propose and test a hybrid algorithm that matches the architecture of the cluster. The scalability of the approach is realized by a domain decomposition method, and the high performance on GPU is realized by using a smoothed aggregation based algebraic multigrid method. Incomplete factorization, which performs beautifully on CPU but poorly on GPU, is completely avoided in the approach. We report some numerical results obtained by using up to 32 CPU/GPU pairs for solving a PDE problem with up to 32 millions unknowns.

1. INTRODUCTION

Many scientific and engineering problems can be studied by solving partial differential equations (PDEs) discretized by a mesh based method such as finite element or finite difference. Mature and general purpose computational algorithms and high performance software are available for CPU-based large scale supercomputers, for example, PETSc [1]. In the past few years, several of the top ranked supercomputers have moved to hybrid architectures consisting of a large number of CPUs and GPUs. Tremendous speedup has been observed, in comparison with CPU-only calculations, for some computational problems with special structures, for example, Fast Fourier Transforms (FFT) on a single GPU card [13] and GPU clusters [3], Fast Multipole Method (FMM) based particle simulations [7, 9, 10]. Although success has been made in solving dense linear algebra problems using GPUs (see, e.g., [18]), most of the general sparse matrix based parallel solvers don't work well on GPUs, because of the unstructured and irregular nature of the problems and, in particular, the poor performance of incomplete factorization algorithms that are often in the inner-most loop of a preconditioned iterative solver. Efforts have been made in exploiting GPU for sparse matrix calculations. For example, the development version [12] of PETSc begins recently to have GPU support via the Cusp [4] and Thrust [17] libraries from NVIDIA. Rocha et. al. [14] implemented a Jacobi-preconditioned conjugate gradient method to solve sparse linear systems arising in cardiac electrophysiology, where both CSR and ELLPACK matrix formats are investigated. More advanced GPU-based preconditioning techniques such as the algebraic multigrid method is employed in [5] and about 100 times speedup is observed on an eight-GPU configuration than a typical server CPU core.

In this paper, in order to avoid the use of incomplete factorization based components in a preconditioner, we propose and test a hybrid algorithm based on a domain decomposition method and an algebraic multigrid method. The basic assumption required by the proposed algorithms is that equal number of CPUs and GPUs are used on each of the computing node in the cluster. The extension of the algorithm to the case of more GPU cards attached to a CPU is straightforward, but has not been studied in this paper. In the algorithm, the partial differential equation is first divided by a partition of the underlying mesh into a number of overlapping submeshes, each is mapped onto a pair of CPU and GPU. Within a computing node, we perform the subdomain preconditioning operation on the GPU and all the other operations on the CPU. To take architectural advantage of the GPU card, we use a smoothed aggregation (SA) based multigrid method which further partitions the

submesh into several much smaller aggregates and the basis of each aggregate gives rise to a set of degrees of freedom on the coarse level. The coarsest level of the SA subdomain preconditioner is solved by a dense LU solver.

The rest of the paper is organized as follows. In Section 2, a hybrid algorithm based on an additive Schwarz preconditioner and an SA subdomain solver is introduced. Numerical results on a NVIDIA Tesla S1070 cluster are then provided in Section 3 to show the efficiency of the proposed method. The paper is concluded in Section 4.

2. A HYBRID ALGORITHM

In many applications, the discretization of a PDE with finite element or finite difference method results in a linear system of equations

$$Ax = b, \tag{1}$$

where A is a large sparse matrix and b a given vector. In this paper, we assume that A is also symmetric positive definite, which is true if, for example, the PDE is a self-adjoint elliptic problem. There are several software packages offering efficient parallel solvers for such problems on supercomputers made of CPUs [1, 6, 8, 19], but algorithms and software that are efficient on a cluster of CPU/GPUs are still lacking. We consider the class of preconditioned iterative methods that solves the preconditioned system

$$M^{-1}Ax = M^{-1}b,$$

where the preconditioner M is an approximation of A^{-1}. We first make two observations:

- The performance of this approach depends heavily on how M^{-1} is defined and implemented, because the computation of $M^{-1}v$ is usually much more expensive than the computation of Av in terms of the compute time, the communication time, and the memory requirement.
- The accuracy of the solution of (1) has (almost) nothing to do with M^{-1}. This means that we have lots of flexibilities about how M^{-1} is computed, and sometimes, we don't have to compute it too accurately in order to obtain higher level of efficiency.

Base on the above observations, we propose to allocate all calculations related to M^{-1} to the GPUs and keep all other calculations on the CPUs. On the GPUs, we approximately compute M^{-1} using a suitable algorithm. Such an approach may not be the best in terms of the total number of floating point operations, but offers much better results in terms of the total compute time.

Because our algorithm is based on domain decomposition and multigrid methods, we further assume that associated with the matrix A, there is a computational domain Ω, with which we obtain mesh based partitions of A. This assumption can be replaced by any graph-based algorithm if the mesh information is not available.

For the model problem studied in this paper, we employ an additive Schwarz preconditioned Conjugate Gradient (CG) algorithm ([15]) to solve (1). The procedure of the preconditioned CG algorithm is provided in Algorithm 1, where M^{-1} is a Schwarz preconditioner.

Algorithm 1 Preconditioned CG for $Ax = b$

1. $r_0 = b - Ax_0$, $z_0 = M^{-1}r_0$, $p_0 = z_0$
2. do $j = 0, 1, ...$ until convergence
   3. $\alpha_j = (r_j, z_j)/(Ap_j, p_j)$
   4. $x_{j+1} = x_j + \alpha_j p_j$
   5. $r_{j+1} = r_j - \alpha_j Ap_j$
   6. $z_{j+1} = M^{-1}r_{j+1}$
   7. $\beta_j = (r_{j+1}, z_{j+1})/(r_j, z_j)$
   8. $p_{j+1} = z_{j+1} + \beta_j p_j$
9. end do

Denote np as the number of CPU/GPU pairs. We partition the computational domain Ω into np non-overlapping subdomains. An overlapping decomposition can be obtained by extending each subdomain with δ mesh layers. Each overlapping subdomain Ω_k is managed by an MPI process assigned to a CPU/GPU pair. The procedure of the additive Schwarz (AS) preconditioner is provided in Algorithm 2, where R_k^T and R_k serve as a restriction operator and an interpolation operator respectively; their detailed definitions can be found in, e.g., [16].

Algorithm 2 Additive Schwarz: $z \leftarrow AS(r)$

1. For $k = 1, 2, ..., np$
   Restriction: $r_k = R_k r$
   Solve the subdomain problem $z_k = B_k^{-1} r_k$
   End for
2. Interpolation: $z \leftarrow \sum_{k=1}^{np} R_k^T r_k$

For each CPU/GPU pair, the subdomain matrix $C = B_k$ and the right-hand side vector $d = r_k$ are both copied to the local memory of the GPU card before solving the subdomain problem on GPU using a SA algorithm described later. Then the solution vector $x = z_k$ on the GPU side is copied back to the local memory of the CPU, which requires synchronization to make sure the global vector is completely assembled. Note that the subdomain matrix is copied to GPU only once and does not need to be copied back. The data between CPU and GPU within each MPI process is typically transfered through the PCI-Express path between the host CPU memory and the GPU memory. A sketch of the additive Schwarz preconditioned CG algorithm is illustrated in Figure 1. We implement the additive Schwarz preconditioned CG algorithm on a cluster of CPU/GPUs, where SA is allocated and executed on the GPUs and all other operations are performed on the CPUs.

We employ a smoothed aggregation (SA) based algebraic multigrid method to solve the subdomain problems B_k^{-1} in the AS preconditioner. The SA algorithm [2] is defined recursively by using several operators. Let P be the prolongation operator which is a full rank matrix whose range contains the algebraically smoothed components of the residual

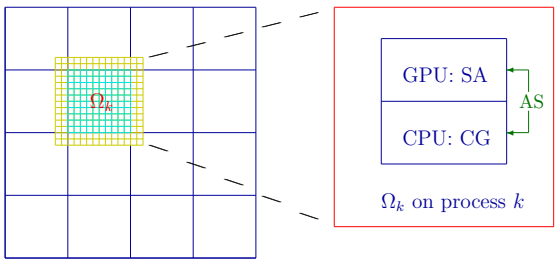


Figure 1: Sketch of the additive Schwarz preconditioned CG algorithm. Left: overlapping domain decomposition of a two-dimensional rectangular domain. Right: Each subdomain is assigned to a MPI process that is further assigned to a CPU/GPU pair.

corresponding to an approximate solution of $Cx = d$, where C and d are the subdomain matrix and the right-hand side for a subdomain problem. With the prolongator, we can define a coarse version of C, as $C_c = P^T CP$, and an iterative method is defined as

$$x \leftarrow x - Py,$$

where y is obtained by solving a coarse grid problem

$$C_c y = P^T(Cx - d).$$

Let $n = n_1$ be the dimension of C, and denote the fine level linear system $Cx = d$ as $C_1 x = d_1$. We introduce a sequence of coarse matrices as

$$C_{l+1} = (I_{l+1}^l)^T C_l I_{l+1}^l,$$

where the prolongator I_{l+1}^l is defined as the product of a given prolongation smoother, S_l, and a tentative prolongator, P_{l+1}^l

$$I_{l+1}^l = S_l P_{l+1}^l$$

for $l = 1, ..., L-1$. One popular choice for the prolongation smoother is Richardson's method:

$$S_l = I - \frac{4}{3\lambda_l} C_l$$

where λ_l is an upper bound on the spectral radius of the matrix on level l. At each level, for the system $C_l x = d_l$, we need a smoother

$$x \leftarrow (IT_l C_l)x + T_l d_l,$$

where T_l is an approximate inverse of C_l for $l = 1, ..., L-1$. Then, SA can be defined as in Algorithm 3.

Since SA is only used as part of a preconditioner, as observed earlier in the paper, the convergence of SA is not necessary. In our implementation, we only apply the smoother for a small number (μ) of sweeps for the best performance in terms of the total compute time. Increasing the number of smoothing steps helps in reducing the total number of outer iterations, but may increase the overall compute time according to our experiments. Between levels, we use either a Jacobi or a polynomial smoother. When the later is used, the basis functions we choose are the Chebyshev polynomials of the first kind. The polynomial of matrices can be computed by a sequence of sparse matrix-vector multiplication (SpMV) that can be applied in a very efficient way.

Algorithm 3 Smoothed Aggregation: $x_l = AMG_l(x_l, d_l)$

0. If on the coarsest level, then:
   Solve $C_l x_l = d_l$ by direct LU, else:
1. Apply μ steps of smoothing to $C_l x = d_l$
2. Coarse grid correction:
   (a). Set $d_{l+1} = (I_{l+1}^l)^T(d_l - Cx_l)$ and $x_{l+1} = 0$
   (b). Solve the coarse problem $B_{l+1}x_{l+1} = d_{l+1}$
   by γ applications of $x_{l+1} = AMG_{l+1}(x_{l+1}, d_{l+1})$
   (c). Then correct the solution on the level l
   by $x_l \leftarrow x_l + I_{l+1}^l x_{l+1}$
3. Apply μ steps of smoothing to $C_l x = d_l$.

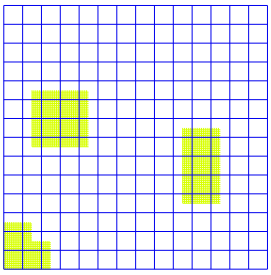


Figure 2: Some possible aggregate candidates on a regular rectangular mesh.

In the SA algorithm, a hierarchy of coarse problems is constructed based on the linear system itself and on certain assumptions about the smooth components of the error. At each level, the prolongation matrix is defined by a decomposition of the set of degrees of freedom associated with the matrix C_l into an aggregate partition, $\{C_l^1, ...C_l^{N_l}\}$ where each aggregate C_l^i is formed based on the connectivity and strength of connection between the elements of C_l, without the need for explicit knowledge of the problem geometry. Figure 2 shows a few possible aggregate candidates on a regular rectangular mesh. The level hierarchy in the SA algorithm is extended until the number of rows in the matrix of the coarsest level is less than 500, which usually results in $4 \sim 5$ levels. Then the matrix on the coarsest level is factorized using a dense LU factorization and is solved by an triangular solver. We employ the MAGMA library[11] which is a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures.

The performance of SA on a GPU depends mainly on three operations: BLAS-axpy (in $2.(a), 2.(c)$), SpMV (in $1., 3.$), and the dense triangular solver ($0.$). On a GPU, thread blocks are assigned to handle rows of the vector associated with some unknowns and the corresponding rows of matrix.

3. NUMERICAL EXPERIMENTS

The experiments were carried out on a NVIDIA S1070 GPU cluster with 14 nodes. Each node is equipped with two quad-core 2.26 GHz Intel Xeon E5520 CPU processors and four 1.3 GHz NVIDIA Tesla C1060 GPU cards. Nodes are interconnected by a 20Gb InfiniBand DDR network. CUDA Toolkit 3.2 are used for programming and the CUDA kernels in the code are compiled by NVIDIA CUDA Compiler with flag `-arch_sm 13` in order to enable double precision. The CPU code is compiled by Intel MPI compiler using `-O3`

optimization level.

In this paper, we study the numerical solution of the Poisson equation with homogeneous Dirichlet boundary conditions on the computational domain $\Omega = [0,1]^2$. A 5-point finite difference scheme is employed to discretize the problem on a uniform $N \times N$ rectangular mesh. The resulting sparse matrix is symmetric positive definite. The stopping condition for the iterative solver is when the relative residual is smaller than 10^{-6}. Even though not tested, we expect the code to work for other second or forth order elliptic problems with variable coefficients.

In the rest of the section, "Iter" is the number of CG iterations, "TSolve" is the total compute time, "TData" is the data transfer time between CPU and GPU, "Eff" is the parallel efficiency as compared with the run using the smallest number of processors in the same table.

First we discuss the parameters of this algorithm and distinguish the optimal. Both the Jacobi and the Chebyshev polynomial smoothers are tested in the experiments. Table 1 shows the impact on the number of iterations and total compute time by using different numbers of sweeps in Jacobi or different degrees of the Chebyshev polynomial. The number of MPI processes is fixed to 32 and the mesh is 8193×4097. The overlap of the additive Schwarz preconditioner is fixed to 1 here. We observe that the number of iterations can be reduced by increasing the number of sweeps of Jacobi or the degree of polynomial of the Chebyshev smoother, however, if the interest is the compute time, 1 sweep of Jacobi is the clear winner.

Table 1: Performance comparison between the Jacobi smoother and the Chebyshev polynomial smoother, with mesh size 8193×4097 **and** $np = 32$**, time is shown in seconds.**

Jacobi smoother			Polynomial smoother		
Sweeps	Iter	TSolve	Degree	Iter	TSolve
1	249	26.687	-	-	-
2	242	28.175	2	306	36.719
3	241	29.579	3	316	39.557
4	240	31.631	4	304	39.489
5	236	33.267	5	285	39.517
6	230	34.093	6	283	40.399

Multiple cycles of SA solves the subdomain problem more accurately, thus results in less iteration count of the outer CG. Table 2 reveals how the iteration count is influenced by the cycles of SA. All cycles are executed on GPU so as to avoid multiple data copies between CPU and GPU. In this test, the Jacobi smoother is used. From the table we see that, as expected, the number of iterations is reduced significantly, and the solving time does not increase until cycles= 3.

We next investigate the optimal size of overlap in the additive Schwarz preconditioner. Generally speaking, a larger overlap usually results in fewer number of iterations due to more communications between subdomains. But the overall compute time may not decrease since the size of the subdomain system, as well as the communication time between the CPUs grow relatively. Table 3 shows the impact of the overlapping size, where the sweeps of Jacobi is 1 and the SA cycles is 3. As shown, it is a little strange that both the number of iterations and the compute time first grow to some extent, then quickly reduce as the overlap increases.

Table 2: Influence by the number of cycles of SA, with mesh size 8193×4097 **and** $np = 32$**, time is shown in seconds.**

Cycles	Iter	TSolve
1	249	26.687
2	209	26.277
3	188	26.919
4	182	29.497
5	178	31.586

Table 3: Impact of overlapping size in the additive Schwarz preconditioner, with mesh size 8193×4097 **and** $np = 32$**, time is shown in seconds.**

Overlap	Iter	TSolve
0	185	25.85
1	188	26.92
2	191	27.71
3	177	26.53
4	161	24.46
5	153	23.78

We then examine the mesh scalability of the hybrid solver by fixing the number of MPI processes to 32 and increasing the mesh size. The results are provided in Table 4, where results using a CPU-based sparse LU factorization (instead of the GPU-based SA method) as subdomain solvers are also included for comparison. The optimal parameters discussed above are used for all tests from now on. The overlap is fixed to 1 since the CPU-based LU approach requires too much memory when the mesh is very fine. It can be seen from Table 4 that when the mesh is small, the hybrid version costs more time than the pure CPU version, but this situation quickly changes when the mesh size grows up to 2049×2049. It is also observed that the number of iterations of the hybrid solver is greater than that of the pure CPU version, due to the fact that SA is unable to solve the subdomain problems as exactly as direct LU. Table 4 also indicates that the time spent on the data transfer (TData) between the CPU and GPU within the same MPI process is almost negligible compared to the total solution time.

Table 4: Performance comparison on the mesh scalabilities between the CPU-based and the hybrid approaches, $np = 32$**, time is shown in seconds.**

	CPU approach		Hybrid approach		
Mesh	Iter	TSolve	Iter	TData	TSolve
513×513	63	0.29	73	0.01	0.67
1025×1025	77	0.77	94	0.04	2.54
2049×2049	98	4.52	126	0.16	3.64
4097×4097	101	24.78	159	0.66	12.68

In the strong scaling test, we use a fixed 2049×2049 mesh and increase the number of MPI processes. In the ideal situation, the compute time should be reduced proportionally as more MPI processes are deployed. Strong scaling results using both the CPU-based and the hybrid one-level approaches are provided in Table 5, from which we see that the hybrid approach is always faster than the CPU-based approach. Superlinear speedup is observed for the CPU-based implementation, but the speedup for the hybrid implementation is not as good.

Table 5: Performance comparison on the strong scalabilities between the CPU-based and the hybrid approaches, mesh size is 2049×2049, time is shown in seconds.

	CPU approach			Hybrid approach		
np	Iter	TSolve	Eff	Iter	TSolve	Eff
2	12	87.37	n/a	62	16.28	n/a
4	55	51.96	84.1%	75	11.77	69.2%
8	79	21.74	100.4%	96	7.88	51.7%
16	75	9.73	112.2%	104	6.79	30.0%
32	98	4.49	121.7%	126	3.57	28.5%

In the weak scaling test, starting from a relatively small 2049×2049 mesh with 4 processes, we increase the number of MPI processes and the mesh size at the same time, so that the mesh size per MPI process is fixed. In the ideal situation, the compute time should remain unchanged which is in fact hard to achieve due to the increasing cost of communication between MPI processes. Table 6 again indicates that the hybrid approach is superior to the CPU-based approach although the parallel efficiency is poor.

Table 6: Performance comparison on the weak scalabilities between the CPU-based and the hybrid approaches, starting from a 2049×2049 mesh with $np = 4$, time is shown in seconds.

	CPU approach			Hybrid approach		
np	Iter	TSolve	Eff	Iter	TSolve	Eff
4	55	51.97	n/a	98	12.09	n/a
8	85	60.09	86%	159	18.40	66%
16	84	60.24	86%	177	20.56	59%
32	114	68.66	76%	188	26.92	45%

4. CONCLUDING REMARKS

In this paper, we proposed and tested a hybrid algorithm based on domain decomposition and smooth aggregation multigrid method for solving elliptic partial differential equations on a cluster of CPUs and GPUs. In the preconditioned Krylov subspace framework, we allocate and execute all preconditioner related operations on the GPUs and all other operations are performed of the CPUs. We carefully investigated the impact of several important parameters that determine the performance of the algorithms. In terms of the number of iterations, the CPU-only approach is clearly better, but for large meshes the hybrid CPU/GPU approach is better in terms of the overall compute time. On the GPU, the mathematically simple Jacobi based smoother performs much better than the more sophisticated Chebyshev polynomial smoother. Our numerical experiments were obtained on a CPU/GPU cluster using up to 32 CPU/GPU pairs, and for a problem with up to 32 millions unknowns.

In this paper, we only considered the case when the number of CPU/GPU pairs in the cluster is relatively small. Multilevel domain decomposition will be necessary for larger clusters. To deal with the additional communication among GPU cards, in multilevel methods, the new feature, GPUDirect, offered in CUDA 4.0 that supports peer-to-peer communication between GPUs over PCIe in the same system, will be very useful. This feature benefits the communication between subdomains in the overlapping Schwarz method, where the input right-hand side and the solution can be sent to the neighboring processors without the aid of CPU memory.

5. ACKNOWLEDGMENTS

We sincerely thank the support of the PETSc group of the Argonne National Laboratory. The work of LL and YBZ was partially supported by the Knowledge Innovation Program of the Chinese Academy of Sciences under grant KJCX2-EW-L01, the work of CY was partially supported by the NSF China grant 61170075, and the work of XCC was partially supported by the NSF grant DMS 0913089.

6. REFERENCES

[1] S. Balay, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang. *PETSc Users Manual*. Argonne National Laboratory, 2010.

[2] M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge. Adaptive smoothed aggregation (α-SA) multigrid. *SIAM Rev.*, 47:317–346, 2005.

[3] Y. Chen, X. Cui, and H. Mei. Large-scale FFT on GPU clusters. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 315–324, New York, NY, USA, 2010. ACM.

[4] The Cusp library. `http://code.google.com/p/cusp-library/`.

[5] C. C. Douglas, H. Lee, G. Haase, M. Liebmann, V. Calo, and N. Collier. Parallel algebraic multigrid method with GP-GPU hardware acceleration. *J. Comput. Appl. Math.* to appear.

[6] R. D. Falgout and U. M. Yang. hypre: A library of high performance preconditioners. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part III*, pages 632–641, London, UK, 2002. Springer-Verlag.

[7] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 62:1–62:12, New York, NY, USA, 2009. ACM.

[8] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Software*,

31:397–423, 2005.

[9] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn. Scaling hierarchical N-body simulations on GPU clusters. In *SC'10: Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[10] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 58:1–58:12, New York, NY, USA, 2009. ACM.

[11] Matrix algebra on GPU and multicore architectures. `http://icl.cs.utk.edu/magma/`.

[12] V. Minden, B. Smith, and M. G. Knepley. Preliminary implementation of PETSc using GPUs. In *Proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering*, 2010.

[13] A. Nukada and S. Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[14] B. M. Rocha, F. O. Campos, R. M. Amorim, G. Plank, R. W. d. Santos, M. Liebmann, and G. Haase. Accelerating cardiac excitation spread simulations using graphics processing units. *Concurr. Comput.: Pract. Exper.*, 23:708–720, 2011.

[15] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS, 1996.

[16] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, New York, 1996.

[17] The Thrust library. `http://code.google.com/p/thrust/`.

[18] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.*, 36:232–240, 2010.

[19] A. M. Wissink, R. D. Hornung, S. R. Kohn, and S. S. Smith. Parallel multi-physics AMR applications using the SAMRAI library. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, page 184, New York, NY, USA, 2001. ACM.

Author Index