Adapting Irregular Computations to Large CPU-GPU Clusters in the MADNESS Framework

Vlad Slavici* Raghu Varier Gene Cooperman*
Northeastern University
Boston, MA
$\{$*vslav,varier,gene*$\}$*@ccs.neu.edu*

Robert J. Harrison
Oak Ridge National Laboratory
Oak Ridge, TN
rjharrison@ornl.gov

Abstract—**Graphics Processing Units (GPUs) are becoming the workhorse of scalable computations. MADNESS is a scientific framework used especially for computational chemistry. Most MADNESS applications use operators that involve many small tensor computations, resulting in a less regular organization of computations on GPUs. A single GPU kernel may have to multiply by hundreds of small square matrices (with fixed dimension ranging from 10 to 28). We demonstrate a scalable CPU-GPU implementation of the MADNESS framework over a 500-node partition on the Titan supercomputer. For this hybrid CPU-GPU implementation, we observe up to a 2.3-times speedup compared to an equivalent CPU-only implementation with 16 cores per node. For smaller matrices, we demonstrate a speedup of 2.2-times by using a custom CUDA kernel rather than a cuBLAS-based kernel.**

I. Introduction

Titan, at Oak Ridge National Laboratory, will likely be the largest supercomputer in the world by the end of 2012 [1]. At that time it will offer up to 18,688 CPU-GPU nodes. Each compute node consists of a 16 core CPU and an NVIDIA Fermi GPU. Titan currently offers 960 CPU-GPU nodes. As of November 2011, 3 of the 10 fastest supercomputers in the world used GPUs [2]. Also, 5 of the world's 10 most power-efficient supercomputers used GPUs [3]. Hence, adapting irregular computations to large clusters of hybrid CPU-GPU nodes is a priority for high-performance computing.

This work focuses on adapting the MADNESS [4], [5], [6], [7], [8] scientific framework to large hybrid CPU-GPU clusters. In an experiment for 4-dimensional tensors on a 500-node partition, we demonstrate a speedup of 1.9 times by using the GPUs of the cluster over using only the CPUs of the cluster. Further, we demonstrate a speedup of 2.3 times over the CPU-only version by dispatching tensor operations to both the CPU and the GPU. In an experiment on the 3-dimensional tensors using 100 nodes, a custom CUDA kernel for tensor operations is used to demonstrate a 1.44-times speedup over an equivalent implementation using the cuBLAS DGEMM routine.

MADNESS (Multiresolution ADaptive Numerical Environment for Scientific Simulation) is a simulation framework that implements a set of basic operations that are used in computational chemistry, nuclear physics and other related fields employing quantum mechanics.

Because other packages do not use adaptive multiresolution methods, they have a tendency to do additional work for the same accuracy. Because the additional work is more regular than that for MADNESS, it tends to parallelize well on GPUs. *Parallelization of MADNESS tends to suffer from many small tasks.* Hence, other packages will achieve higher speedups on GPUs because they begin with a lower performance on CPUs for a given accuracy.

> *Hence, for a given accuracy, MADNESS already achieves good performance on CPU-based clusters. The challenge for MADNESS is to achieve enough speedup on GPUs so as to retain its performance advantage for a given accuracy.*

Most applications running on MADNESS use 3- and 4-dimensional tensors[1]. The four most common operators in MADNESS are `Apply`, `Compress`, `Reconstruct` and `Truncate`. These four operators suffice for many MADNESS applications. Only `Apply` is CPU-intensive and the time for calls to `Apply` dominates over the other three.

The `Apply` operator involves many small tensor multiplications often running into the hundreds. This results in a less regular organization of computations on GPUs. CUDA kernels that perform many tensor operations without leaving the GPU are the key to obtaining high performance for these less regular computations.

A naive CPU-GPU port of a MADNESS operator would replace each call to a matrix multiplication on the CPU with an equivalent GPU routine. However, this would result in low GPU occupancy and high CPU-GPU transfer latency. Also, CPU-GPU transfer would optimally use page-locked memory, but the overhead of page-locking for the transfer of a single matrix would be excessive. The solution is:

- to separate compute-intensive code from data-intensive code;
- to aggregate the computation; and

* This work was partially supported by the National Science Foundation under Grant CCF 0916133.

[1] Tensors are higher dimensional generalizations of standard 2-dimensional matrices.

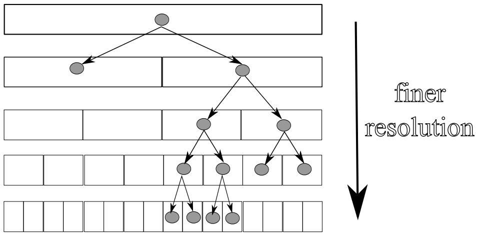

Figure 1: The MRA approach can be viewed as a telescoping series of grids. Grid V_0 is the coarsest. The finer the grid, the more information is captured about the wave function in the specific area. The more irregular the function in a certain area, the finer the grid that is necessary to represent it.

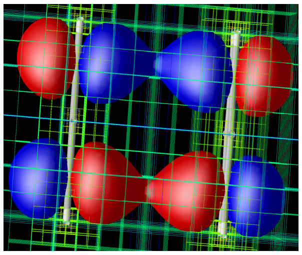

Figure 2: Multiresolution Analysis over benzene dimer.

- to aggregate the data inputs.

Due to the irregular nature of the computations, even hand optimization of the MADNESS operators for the GPU does not utilize the full power of the GPU. Hence, it was important:

- to overlap CPU computation with GPU computation.

MADNESS on a cluster already efficiently handles communications between compute nodes and Titan does not introduce additional bottlenecks.

A. *MADNESS — a Scientific Simulation Framework based on Multiresolution Analysis*

This work focuses on adapting the MADNESS [4], [5], [6], [7], [8] scientific framework to hybrid CPU-GPU clusters.

MADNESS employs a Multiresolution Analysis (MRA) methodology to reduce the computational complexity of the problem and achieves high precision by performing an adaptive mesh refinement over the simulation volume. The different levels of refinement have different accuracy and the number of levels depends on the precision (accuracy) requested by the user (see Figure 1).

Figure 2 shows grids having finer resolution over the regions of a molecule where the probability of finding electrons (electron density) is higher and have coarser resolution over empty regions.

This helps in concentrating most of the compute power towards relevant computations, thereby reducing overall computational complexity. The level of refinement is improved by progressively generating new levels using the previous levels until the desired accuracy is achieved.

A multiresolution grid is represented in MADNESS as a highly unbalanced tree (see Figure 1). The nodes of the tree are distributed across the nodes of a cluster. The distribution is done using a tree-node to compute-node mapping. There are much more tree-nodes than compute-nodes and a tree-node resides on a single compute-node. Distributed trees are implemented in MADNESS with distributed hash tables.

MADNESS operators (such as `Apply`, `Compress`, `Reconstruct`, or `Truncate`) take as input a distributed tree, which they explore and modify.

B. *Adapting Scientific Simulation Frameworks for CPU-GPU clusters*

Among packages supporting molecular dynamics, MADNESS is unique in using adaptive multiresolution analysis to solve molecular dynamics problems larger than ever before. The adaptively refined tree (see Figure 1) leads to computations at multiple scales. This involves irregular computations over small matrices.

The multiple levels and scales for MADNESS are one of its strengths. The only competitive approach is BigDFT [9]. However, BigDFT only has two levels of resolution (coarse and fine). This makes it significantly less flexible than MADNESS. BigDFT performs a higher amount of computation and uses much larger matrices for similar problems, compared to MADNESS.

Some of the competing scientific simulation frameworks that have been ported to CPU-GPU architectures are [10]: BigDFT (a pseudopotential density functional theory (DFT) framework that expresses Kohn-Sham wavefunctions in Daubechies wavelet bases), TeraChem (Gaussian orbitals that implement ab initio molecular dynamics and DFT methods), GAMESS (also an ab initio computational chemistry framework), AMBER (a framework for classic molecular dynamics), NAMD (molecular dynamics), GROMACS (molecular dynamics for protein and bio-molecule simulation), LAMMPS (classic molecular dynamics) and QMCPACK (continuum quantum Monte Carlo methods).

C. *Organization of Paper*

Section II presents our extensions to the MADNESS Library to support CPU-GPU clusters. Experimental results are presented in Section III. Related work relevant to hybrid CPU-GPU cluster computing is presented in Section IV. Section VI presents our conclusions and discusses future work.

II. Extending MADNESS to CPU-GPU Clusters

Extending MADNESS to CPU-GPU Clusters requires modifications to the underlying control flow of MADNESS operators. Keeping the current control flow for a CPU-GPU execution would be inefficient, since launching a GPU kernel on-demand is slow, because of: high CPU-GPU transfer latency, low CPU-GPU transfer bandwidth without page-locking, slow on-demand page-locking, and low GPU occupancy. Our modified MADNESS operators are fully compatible with the rest of the framework, that was not modified.

A. Modifications to the Control Flow of a MADNESS Operator

As described in the introduction, MADNESS employs many small tasks as opposed to a few large tasks. One MADNESS task applies an operator to a single node in the multiresolution tree and it can pass the result to other tree-nodes, modify the current tree-node, remove it and even create new tree-nodes. A node of an n-ary tree consists of an n-dimensional tensor along with extra information.

The most important modification made to the control flow of MADNESS operators for CPU-GPU execution is the *asynchronous batching* of MADNESS tasks and task inputs. Asynchronous batching is critical for solving the high latency, slow transfer and GPU occupancy issues.

Asynchronous Batching of Data: Asynchronous batching enables the aggregation of data inputs for GPU tasks. Data inputs are aggregated into a few large pre-allocated buffers, which are then transferred to the GPU in a single step. The CPU-GPU latency penalty is thus paid only once for the entire batch as opposed to once for each task input.

Moreover, the pre-allocated transfer buffers are page-locked at the beginning of the computation. Page-locking ensures that the buffer stays resident in RAM, and it also leads to at least double the transfer speed. Page-locking can efficiently be done only on a few large buffers, since it is slow (0.5 milliseconds). Page-unlocking (the reverse of page-locking) is even slower (2 milliseconds), in the context in which the execution of a single typical MADNESS 3-dimensional CUDA kernel is on the order of 1 millisecond.

Asynchronous Batching of Computation: Besides input data, the compute-intensive MADNESS tasks also are asynchronously batched. A batch of tasks can be more efficiently scheduled on the streaming processors (SMs) of the GPU than individual tasks. Having a batch of tasks readily available allows launching multiple tasks of the same kind on the GPU concurrently in the case of small tasks, that individually occupy only a fraction of the GPU.

Moreover, informed decisions can be made about how to divide work between the CPU and the GPU when a large computation batch is available.

MADNESS Library Extensions for Asynchronous Batching: It is not desirable to perform batching of all MADNESS tasks. The goal is to batch only those MADNESS tasks that are compute-intensive. The MADNESS algorithms developer has to identify these tasks and expose them to the MADNESS Library extensions. Specifically, the developer can split a task of interest into three sub-tasks: *preprocess*, *compute* and *postprocess*. The MADNESS Library extensions will ensure that the *preprocess* sub-task will be executed by a CPU thread. It will also ensure that the output data of *preprocess* is batched together with other output data of the same kind, to serve as input data for *compute* tasks.

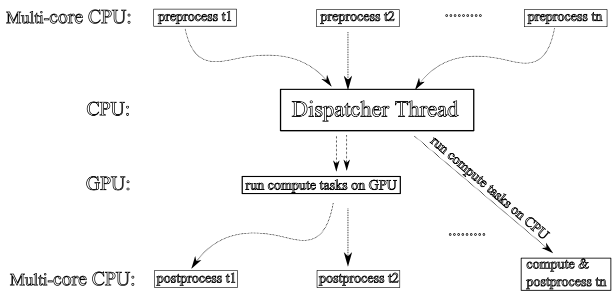

Figure 3: The control flow of a hybrid CPU-GPU MADNESS operator.

The execution of the multiple *compute* tasks waiting for input data is delayed until a timer expires. At this point there are multiple batches of *compute* waiting to be executed (one batch per kind[2] of *compute* task). Batches of *compute* tasks will be executed one by one at this point.

A dispatcher CPU thread will split each batch of *compute* tasks between the CPU threads and the GPU. A *compute* task must have both a CPU- and a GPU-version. By knowing the relative performance of the GPU code compared to the CPU code for a certain operator, a MADNESS developer can decide what is the ratio of CPU to GPU work. Consider that a CPU-only run takes time m and a GPU-only run takes time n. The minimal computation time can be achieved by an optimal CPU-GPU computation overlap. The *optimal overlap* occurs when the CPU and GPU remain fully loaded throughout. The minimal time is calculated by minimizing $max(mk, n(1-k))$, with $k \in [0, 1]$. This means that a k-fraction of tasks are sent to the CPU and the rest to the GPU. The optimal CPU-GPU work overlap is achieved when $mk = n(1-k)$, so $k = n/(m+n)$. The minimal runtime is thus $\frac{m \times n}{m+n}$.

Newly generated *compute* tasks continue to be batched while the CPU and the GPU do work. The dispatcher also drives the execution of the GPU computation batches.

The control flow of a MADNESS operator that uses asynchronous batching for hybrid CPU-GPU execution is presented in Figure 3.

Next we present the CPU-GPU version of the MADNESS `Apply` operator.

B. The `Apply` Operator

The most computationally intensive operator in MADNESS is `Apply`. It consists of computing a Gaussian opera-

[2] The "kind" of a task is given by a combination of the memory address of the *compute* function and the result of a user-defined hash function applied to the input data.

tor on each tensor node in a tensor tree and accumulating the local results of the Gaussian to compute an approximation of a version of Green's function.

The computational part of the `Apply` operator can be expressed as:

Formula 1: Integral Kernel

$$r_{i_1 i_2 \dots i_d} = \sum_{\mu=1}^{M} \sum_{j_1=0}^{2k-1} \sum_{j_2=0}^{2k-1} \cdots \sum_{j_d=0}^{2k-1} s_{j_1 j_2 \dots j_d} h^{(\mu,1)}_{j_1 i_1} h^{(\mu,2)}_{j_2 i_2} \dots h^{(\mu,d)}_{j_d i_d}$$

Here r and s are output/input d-dimensional tensors, respectively. s is a tensor from the input MADNESS multiresolution tree, while r is a tensor that will be used to update the input tree. The h operators are 2-dimensional tensors that are either computed as needed, or obtained from a software cache. Typical values of M and k are 100 and 10–20, respectively.

Algorithm 1 describes the MADNESS `Apply` algorithm for CPUs at a high level. Here we are interested in identifying the data-intensive and compute-intensive parts of the code, in order to reorganize them for efficient CPU-GPU execution.

Algorithm 1 The "Apply" Algorithm

```
Input: The current coefficients tree.
Output: The coefficients tree after computing a Green's convolution.
for each node $s_{j_1 j_2 \dots j_d}$ in the coefficients tree do
    Obtain displacements.
    for each displacement do
        neighbor = Compute neighbor of $s_{j_1 j_2 \dots j_d}$ based on displacement.
        Tensor $r_{i_1 i_2 \dots i_d}$ = integral_operator($s_{j_1 j_2 \dots j_d}$).
        Accumulate tensor $r_{i_1 i_2 \dots i_d}$ into neighbor.
    end for
end for
```

Algorithm 2 FUNCTION integral_operator ($s_{j_1 j_2 \dots j_d}$)

```
Initialize result tensor $r_{i_1 i_2 \dots i_d}$.
for each $\mu = 0$ to convolution rank do
    Obtain the $h$ 2-D tensors ($h^{(\mu,1)}_{j_1 i_1}$, $h^{(\mu,2)}_{j_2 i_2}$, ... $h^{(\mu,d)}_{j_d i_d}$).
    Apply Formula 1 to $s_{j_1 j_2 \dots j_d}$ and the $h$ tensors and add the result to $r_{i_1 i_2 \dots i_d}$.
end for
return $r_{i_1 i_2 \dots i_d}$.
```

To take advantage of the MADNESS Library extensions for CPU-GPU, the `Apply` operator presented in Algorithm 1 has to be split into three parts that are implemented by three tasks: *preprocess*, *compute* and *postprocess*.

The *preprocess* task obtains the addresses of all the 2-dimensional h tensor operators. The *compute* task uses the preprocessed inputs, performs the necessary $s \times h$ tensor products and adds the results into tensor r. The *postprocess* task accumulates the result tensor r into a neighbor tensor in the tree.

The bulk of the data is comprised of the two-dimensional tensor operators h. Many of the 2-D tensor operators will be reused multiple times by transformations applied to various different input d-dimensional tensors. Therefore, in order to avoid redundant data transfers to the GPU, a write-once software cache containing the already transferred 2-D tensors has been implemented. This write-once cache has been modeled after a CPU software cache present in MADNESS for similar purposes.

Algorithm 3 The CPU-GPU Version of the "Apply" Algorithm

```
Input: The current coefficients tree.
Output: The coefficients tree after computing a Green's convolution.
for each node $s_{j_1 j_2 \dots j_d}$ in the coefficients tree do
    Obtain displacements.
    for each displacement disp do
        Asynchronously call integral_preprocess($s_{j_1 j_2 \dots j_d}$, disp).
    end for
end for
```

Algorithm 4 FUNCTION integral_preprocess(*source_tensor*, *displacement*)

```
neighbor = Compute neighbor of source_tensor based on displacement.
for each $\mu = 0$ to convolution rank do
    Obtain the $h$ 2-D tensors ($h^{(\mu,1)}_{j_1 i_1}$, $h^{(\mu,2)}_{j_2 i_2}$, ... $h^{(\mu,d)}_{j_d i_d}$).
end for
Asynchronously call integral_compute(source_tensor, neighbor, $h$ tensors).
```

Algorithm 5 FUNCTION integral_compute(*source_tensor*, *neighbor*, h tensor)

```
Initialize result tensor $r_{i_1 i_2 \dots i_d}$.
for each $\mu = 0$ to convolution rank do
    Apply Formula 1 to $s_{j_1 j_2 \dots j_d}$ and the $h$ tensors and add the result to $r_{i_1 i_2 \dots i_d}$.
end for
Asynchronously call integral_postprocess($r_{i_1 i_2 \dots i_d}$, neighbor).
```

Algorithm 6 FUNCTION integral_postprocess(*result*, *neighbor*)

```
Accumulate tensor $r_{i_1 i_2 \dots i_d}$ into neighbor.
```

After all the necessary data has been transferred to or located on the GPU, independent CUDA computational kernels are launched on the NVIDIA device in separate CUDA streams. For 3-dimensional tensors, the use of CUDA streams helps occupy the GPU fully.

The hybrid CPU-GPU version of `Apply`, based on asynchronous batching, is presented in Algorithm 3.

C. CUDA Kernels for `Apply`

We present custom CUDA kernels that take advantage of shared memory, L1 and L2 cache locality, and register locality on NVIDIA GPU devices, and employ coalesced memory access.

For small 3-D tensors the custom CUDA kernels use only two or three Streaming Multiprocessors (SMs) of the 16 available SMs on a high-end NVIDIA Fermi device. CUDA streams are used to concurrently launch five to eight independent CUDA kernels.

The efficiency of the custom CUDA kernels lies mainly in the fact that many computational steps are embedded within the kernels. A traditional approach would implement these computational steps by launching a separate matrix multiplication kernel (provided by a linear algebra library, such as cuBLAS) for each step. However, launching a separate kernel for each computational step cannot take advantage of shared memory locality, since all locality is lost between two consecutive calls. Also, the CUDA kernel launch overhead is an issue, since for small matrix multiplications there is too little computation to hide the kernel launch overhead. Instead, by embedding several computational steps within a single CUDA kernel we alleviate these issues and obtain better performance (see the results in Tables III and IV).

The h tensors in Formula 1 are stored as 2-dimensional small square matrices, while the s n-dimensional tensor is stored as a highly rectangular 2-dimensional matrix. Formula 1 is highly computationally intensive. The amount of necessary computation can be reduced by exploiting some properties of the h matrices.

For 3-dimensional tensors the CPU implementation of matrix multiplication is highly efficient, achieving up to 6 GFLOPS on a single core. For higher-dimensional tensors the CPU implementation is less efficient, since tensors overflow L2 cache.

Our custom CUDA kernel that implements Formula 1 is described in Algorithm 7.

Algorithm 7 The Custom CUDA Kernel for Formula 1

```
Input: The $s$, $h^{(\mu,d)}_{j_d i_d}$ tensors and rank($M$) described in Formula 1.
Output: The result tensor $r$.
Initialize result $r$.
for each $mu < rank$ do
    for each $d < NDIM$ do
        Load $s$ from global memory in coalesced manner.
        Load the current $h$ tensor into shared memory in coalesced manner.
        Perform an optimized matrix multiplication.
        Save the result of the multiplication in shared memory.
        Write the result to global memory in coalesced manner.
        Interblock synchronization using atomic ops.
    end for
    $r = r + s$.
end for
```

Our custom CUDA kernel uses only a small number of thread blocks for the 3-dimensional case (two or three thread blocks). Each thread block achieves good occupancy of one SM and uses all the resources of the single SM. This prevents two thread blocks from being scheduled on the same SM concurrently. As a consequence, an inter-block synchronization mechanism (originally developed by Xiao and Feng [11]) is used as a barrier mechanism for all thread blocks of the same CUDA kernel. Each instance of the custom CUDA kernel is run by a CUDA stream, to achieve task parallelism on the GPU.

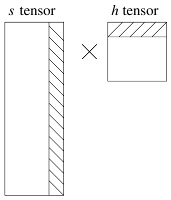

Figure 4: Rank reduction for an s and h tensor. h is a 2-dimensional tensor. s is a higher-dimensional tensor projected onto 2 dimensions. The hashed areas mark the rows and columns that are reduced. Note that reducing the rows and columns does not change the dimension of the result matrix.

D. CPU Optimization: Rank Reduction

MADNESS introduces a separated representation that speeds up higher-dimensional computations, but also expands the rank [8]. For this reason, some of the h tensors in Formula 1 can be approximated by matrices of lower rank. MADNESS implements a CPU optimization in which, for each $s \times h$ multiplication, certain rows and columns of s and h can be omitted (see Figure 4). This optimization is called *rank reduction*. This MADNESS optimization can reduce the amount of computation *on the CPU only* by up to 2.5-times in typical cases.

Rank reduction was also implemented for the custom CUDA kernel, but did not have a noticeable effect on performance. The reason is that, unlike for the CPU, GPU resources are allocated at CUDA kernel launch time. Our custom CUDA kernel typically performs hundreds of multiplications using the same input s tensor, but hundreds of input h tensors, as needed by Formula 1. (Recall that typical values of M and k are 100 and 10–20, respectively, in Formula 1.) Each multiplication uses two or three SMs, due to the lack of enough storage for the computation in the shared-memory and registers of a single SM. The custom kernel must reserve in advance the two or three SMs. For some of the multiplications, rank reduction allows the multiplication to be computed by a single SM. However, the GPU gains nothing from this, since the two or three SMs were already reserved by the kernel.

The dynamic parallelism featured in the future CUDA 5 release could help alleviate some of the rank reduction issues on GPUs. This future facility allowing the launch of subkernels from running kernels seems the most helpful for rank reduction. However, this will only be available for the Kepler GPU.

III. Experimental Results

Experiments were run on nodes of the Titan supercomputer at Oak Ridge National Laboratory. One compute node consists of a 16-core AMD Opteron 6200 Interlagos at 2 GHz frequency, 16 or 32 GB of DDR3 RAM and an NVIDIA Tesla M2090 (of the Fermi class) at 1.6 GHz with 6 GB of GDDR5 connected via a PCIe 2.0 × 16 slot.

One of the applications that relies on `Apply` is the computation of a *Coulomb* operator. The CPU-GPU implementation of `Apply` was compared with a highly-optimized CPU implementation (in which tensor multiplications were programmed in assembly language to achieve good cache behavior). The *Coulomb* application has among the inputs the dimension of the input tensors (d), the size of the tensor per dimension (k) and the desired precision of the result.

Tables I and II present a comparison between the running times of a CPU computation, a GPU computation, and a hybrid CPU-GPU computation of the *Coulomb* operator. The GPU and hybrid CPU-GPU versions both used our MADNESS Library extensions that automatically schedule tasks on the GPU and CPU. These results are observed for a computation batch of 60 independent tasks. Also note that our approach of dedicating two or three SMs on the GPU to one lengthy, compute-intensive task that multiplies many small tensors, is more effective than using cuBLAS (see Tables III and IV) for 3-dimensional problems. cuBLAS distributes a tensor product across all 16 SMs of the GPU, which is efficient for operators on larger tensor sizes, such as the 4-dimensional TDSE operator presented in Table VI.

Application Coulomb with input parameters $d = 3$, $k = 10$ and precision 10^{-8} (no rank reduction)					
CPU-only compute		**GPU-only compute**		**CPU and GPU compute**	
CPU threads	CPU time (sec)	GPU streams	GPU time (sec)	CPU + GPU time (sec) using 10 CPU threads & 5 CUDA streams	
				Actual	Optimal CPU-GPU Overlap
1	132.5	1	71.3	14.4	12.1
2	66.5	2	41.5		
4	45.7	3	31.5		
6	35.6	4	26.4		
8	28.5	5	24.3		
10	24.3	6	24.7		
12	22.8				
14	18.5				
16	19.9				

Table I: CPU scale-up vs. GPU scale-up for *Coulomb* with input parameters $d = 3$, $k = 10$ and precision 10^{-8}. The GPU and hybrid CPU-GPU versions used our MADNESS Library extensions for work scheduling. For the GPU version 12 CPU threads for data access were used.

Table II shows experimental results for running *Coulomb* with $d = 3$, $k = 20$ and precision=10^{-10}. Note that, compared to the $k = 10$ case, here the GPU performs even better compared to the CPU. The larger the tensor size, the better the GPU fares compared to the CPU. The main reason is that the CPU incurs more cache misses for larger tensor sizes. Also, in this case cuBLAS routines were used — with tensors 8 times larger than for $k = 10$, we enter the regime in which cuBLAS performs well.

Tables III and IV compare the performance of a version of the 3-dimensional *Coulomb* application that uses our custom CUDA kernels with a version that uses cuBLAS 4.1. In both versions the computationally intensive part is processed only by the GPU. For this test only we use a MADNESS process map that distributes work evenly among all compute nodes.

Table V presents the running times of a slightly larger *Coulomb* application. In this case 3-D tensors with $k = 30$ were used and the desired result precision was set to 10^{-12}. This application stops scaling above six compute nodes, because there is not enough work. Also, in this case MADNESS does not distribute work evenly between compute nodes, but rather attempts to achieve work locality on compute nodes depending on the shape of the highly unbalanced tree. The work distribution in MADNESS is done according to a process map specification.

Experimental results for a much larger application (a 4-dimensional Time-Dependent Schrodinger Equation — TDSE) are presented in Table VI.

Next we compare our custom CUDA matrix multiplication with the cuBLAS 4.1 matrix multiplication routine. This comparison only was performed on a 16-core Intel Xeon X5570 with a locally attached NVIDIA GeForce GTX 480. Figure 5 compares the custom kernel with cuBLAS for a batch of matrix multiplications corresponding to 3-dimensional tensor products. The k in the figure is the size of the tensor in one dimension. All multiplications for 3 dimensions are of a (k^2, k) matrix × a (k, k) matrix. Fig-

Application Coulomb (no rank reduction) with $d = 3$, $k = 20$ and precision 10^{-10}	
CPU 16 threads time	173.3 sec
GPU time	136.6 sec
CPU + GPU time (actual)	99.0 sec
CPU + GPU time (optimal CPU-GPU overlap)	76.2 sec

Table II: CPU 16 threads vs. GPU for *Coulomb* with input parameters $d = 3$, $k = 20$ and precision 10^{-10}. The GPU-version used 15 CPU threads for data access. The hybrid version used 15 CPU threads.

	Application Coulomb (no rank reduction) with $d = 3$, $k = 10$ and precision 10^{-10}		
Compute nodes	Time (seconds)		Speedup ratio
	Custom kernel	**cuBLAS version 4.1**	
2	88	247	2.80
4	56	126	2.25
8	31	71	2.29
16	19	42	2.21

Table III: Timings for 3-dimensional *Coulomb* using our custom CUDA kernels and using cuBLAS 4.1. Work was distributed evenly to all compute nodes. Below 2 nodes the data per node is too large for the GPU RAM. Above 16 nodes the amount of work in a batch of tasks is insufficient for good parallelism.

	Application Coulomb (no rank reduction) with $d = 3$, $k = 10$ and precision 10^{-11}		
Compute nodes	Time (seconds)		Speedup ratio
	Custom kernel	**cuBLAS version 4.1**	
16	27.6	43.2	1.56
32	15	24.2	1.61
64	10.2	15.6	1.52
100	7.6	11	1.44

Table IV: Timings for 3-dimensional *Coulomb* using our custom CUDA kernels and using cuBLAS 4.1. Work was distributed evenly to all compute nodes. Below 16 nodes the data per node is too large for the GPU RAM. Above 100 nodes the amount of work in a batch of tasks is insufficient for good parallelism.

	Application Coulomb with input parameters $d = 3$, $k = 30$ and precision 10^{-12}				
	Time (seconds)				
Compute nodes	**CPU-only compute**		**GPU-only compute**	**CPU-GPU compute no rank red.**	
	rank red.	**no rank red.**		Actual	Optimal CPU-GPU Overlap
1	147	447	212	172	144
2	115	299	90	60	69
4	114	234	55	39	45
6	96	201	35	25	30
8	102	205	37	25	31

Table V: CPU-only, GPU-only and hybrid computation scale-up with the increase in number of compute nodes. 3-D tensors with $k = 30$ are used. The desired precision is set to 10^{-12}. In the presented results, the CPU-only compute version uses 16 threads, while the GPU-only compute and hybrid versions use 6 CUDA streams and 15 CPU threads. The CPU-GPU dispatcher thread is also active. For the hybrid CPU-GPU case, the CPU-only compute and GPU-only compute times were taken into account in order to divide work optimally between the CPU and the GPU.

	4-D Time-Dependent Schrodinger Equation $k = 14$, precision 10^{-14} (with rank reduction)				
	Time (seconds)				
Compute nodes	**CPU-only compute**	**GPU-only compute** (using cuBLAS)	**CPU-GPU compute**		Speedup of CPU-GPU version over CPU-only version
			Actual	Optimal CPU-GPU Overlap	
100	985	873	664	463	1.4
200	759	580	524	329	1.4
300	739	533	308	310	2.3
400	718	448	299	276	2.4
500	648	339	277	223	2.3

Table VI: Timings for the `Apply` part of the MADNESS 4-dimensional Time Dependent Schrodinger Equation (TDSE) for $k = 14$ and threshold 10^{-14} on Titan. Various runs used between 9 and 14 CPU threads. Observations showed no significant scale-up when using more than 9 threads, so the variable number of CPU threads used does not impact reported results.

ure 6 compares the custom kernel with cuBLAS for a batch of matrix multiplications corresponding to 4-dimensional tensor products. In this second case the multiplications are of a (k^3, k) matrix with a (k, k) matrix.

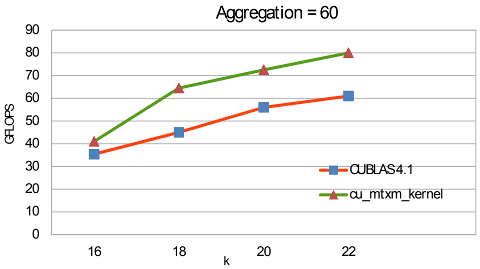

Figure 5: Matrix multiplications corresponding to 3-dimensional tensor products. Measured performance (in GFLOPS) is for batches of 60 multiplications. Higher is better. Each matrix multiplication is $(k^2, k) \times (k, k)$. Our custom kernel is denoted by cu_mtxm_kernel.

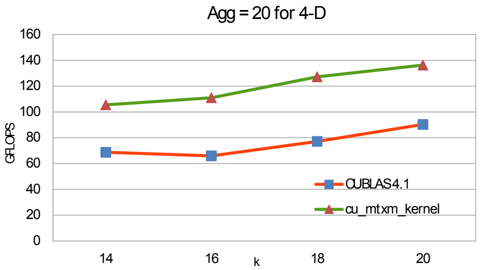

Figure 6: Matrix multiplications corresponding to 4-dimensional tensor products. Measured performance (in GFLOPS) is for batches of 20 multiplications. Higher is better. Each matrix multiplication is $(k^3, k) \times (k, k)$. Our custom kernel is denoted by cu_mtxm_kernel.

A. *Analysis of Results*

Table III presents results for running a *Coulomb* application with precision 10^{-10}. This application scales well up to 16 nodes. Above this threshold there are too few tasks for good load balancing. MADNESS uses static load balancing. To scale above 16 nodes, a larger *Coulomb* application (with precision 10^{-11}) was needed (see Table IV). This second application consists of 154,468 tasks, and it scales up to 100 nodes. Larger applications would scale beyond 100 nodes. The 4-dimensional TDSE application (see Table VI) scales up to 500 nodes. It consists of 542,113 tasks, but these tasks have more computation than the tasks for the 3-dimensional *Coulomb* application, since the matrices are 2-dimensional projections of 4-dimensional tensors.

Note that the speedup of the computation with respect to the number of compute nodes in Table V is not linear since work is not distributed evenly to all compute nodes. MADNESS uses the concept of a process map to specify the distribution of tasks to nodes. In these tests (except Table III and Table IV), the process map assigns more work to some of the nodes. Note that there is no speedup from 6 to 8 compute nodes. Under the current process map there is not enough work to distribute to 8 compute nodes in this test.

Also, for both 6 and 8 compute nodes, the theoretical optimal CPU-GPU computation overlap, calculated taking into account the time for CPU-only compute and GPU-only compute, is higher than the actual time obtained. This can happen because the formula for calculating the optimal CPU-GPU overlap considers the applications to be 100% compute-intensive. However, this is not the case in practice: the tested applications also have data-intensive parts that account for a non-dominating, but still significant fraction of the running time.

An issue for the CPU-only version for larger tensors (as in the case of Table V) is that the computation is saturated by 10 threads, because the working set size is much larger than 16 MB, which is the aggregate size of the L2 cache on the compute nodes of Titan.

The same statement is true for the CPU-only version of an even larger computation, the 4-dimensional Time Dependent

Schrodinger Equation (TDSE).

A 4-dimensional TDSE computation (see Table VI) requires hundreds of compute nodes. For the operations with larger tensors employed in this application we used cuBLAS, since this is the regime in which cuBLAS performs well (large matrix multiplications). Aside from custom CUDA kernels, all other MADNESS extensions were used for these tests. As discussed before, the scale-up is not linear, because of the way MADNESS distributes work using process maps.

Table VI shows that the GPU version scales better than the CPU version. The reason is that some compute nodes are assigned too few tasks at one time to constantly keep all the CPU cores busy. Currently there is no MADNESS CPU implementation of multiple threads working on the same multiplication, whereas for the GPU there is (by using cuBLAS). Therefore, by using the GPU we can efficiently perform large multiplications that are slow on the CPU. The overlapping of the CPU and GPU computation yields good results for 4-dimensional TDSE, as presented in Table VI.

Once again we notice the "super-optimal" overlap of CPU and GPU computation in Table VI, for the case of 300 nodes. As explained before, the reason is the contribution of the data-intensive part of the computation, which is not taken into account when the work division between the CPU and the GPU is performed.

The CPU, besides computation, also has to run all *preprocess* and *postprocess* tasks, which are heavily data-intensive, thus incurring a penalty besides computation. In addition, the dispatcher CPU thread has to rearrange and batch data for the GPU, which also incurs an extra overhead.

IV. Related Work

There has been significant related work that addresses both aspects (code reorganization and GPU execution) of migrating an existing HPC scientific framework to hybrid CPU-GPU computing.

As the two migration aspects are orthogonal, we present related work relevant to each of the aspects.

A. Code Reorganization for Hybrid CPU-GPU Architectures

MapReduce-like Programming Models for Hybrid CPU-GPU: A number of MapReduce frameworks have been proposed for GPU programming in recent years: for NVIDA GPUs He et al. proposed Mars [12] and Catanzaro et al. proposed [13], while for AMD GPUs Elteir et al. proposed [14]. These frameworks only address designing the GPU computation phase as a MapReduce operation, and do not provide solutions for hybrid CPU-GPU computing, such as reorganizing the computation so that it leverages both the CPU and GPU and so that it fits with an existing programming environment.

There are, however, MapReduce-like frameworks for hybrid CPU-GPU computations, such as MapCG [15] or GPMR [16]. Users can write a MapCG or GPMR program and the framework will translate it into either a many-core CPU implementation, a GPU implementation, or a hybrid CPU-GPU implementation. Both MapCG and GPMR have mostly been employed for applications that heavily favor either the CPU or the GPU. In our case, we are looking at applications that do not heavily favor either the host CPU or the device GPU.

Dataflow Graphs for Hybrid CPU-GPU: An interesting approach to providing a high-level programming model for hybrid CPU-GPU systems is CnC-CUDA [17], a declarative deterministic coordination language that is an extension of Intel's Concurrent Collections (CnC) [18]. Programming in CnC-CUDA involves expressing the computation as a static graph in which the nodes can be dynamic computation items, data items and control items. While this approach may be efficient for some algorithms, it is too limiting for complex computations. The structure of such computations cannot easily be described in a static graph.

Closely related to our work is the SkePU [19] skeleton-based framework for hybrid CPU-GPU computing. SkePU provides a set of complex programming skeletons, that are well-suited for GPU constructs such as *Map*, *Reduce*, *MapReduce*, *MapOverlap*, *MapArray*, *Scan* (for data parallelism), and *Farm* (for task parallelism) SkePU can be used in conjunction with StarPU [20] (a runtime system for heterogeneous multicore platforms). However, there is no built-in support for aggregating tasks of the "same kind", which is an important feature for both task scheduling and delayed data access, as needed in this work.

General heterogeneous computing frameworks, such as Harmony [21] or StarPU [20], have a scheduler at their core. The scheduler can use framework-provided scheduling policies, or users can implement their own scheduling policies. These frameworks can be used together with the work presented here.

B. High-level Solutions for obtaining Efficient GPU code

Writing efficient GPU code is a cumbersome task. For linear algebra, libraries such as cuBLAS have been developed. We have found that cuBLAS is efficient for some irregular kernels, as long as the computation to data access ratio is not very small. When that ratio is very small (< 30) custom kernels seem to be the only solution that results in acceptable performance.

Recent hybrid computing solutions such as the CAPS Hybrid Multi-core Parallel Programming (HMPP) [22], OpenACC [23] or the OpenMP extensions of [24] rely on the developer inserting compiler directives in the code for GPU processing. Both HMPP and OpenACC have been inspired by OpenMP [25], a compiler directive approach to multithreaded CPU computing. As of now, they cannot compete with custom approaches for irregular kernels.

V. Acknowledgment

We wish to thank Judith C. Hill for some test cases and for valuable assistance in using Titan.

VI. Conclusion and Future Work

Scalability of an irregular computation was demonstrated through the level of 500 hybrid CPU-GPU nodes. The key was separation of the work into data-intensive routines on the CPU and compute-intensive tasks on the GPU. The data was aggregated for efficient transfer between CPU and GPU. Multiple kernels were streamed to the GPU, with the GPU executing 5 streams at once. Simultaneously, each of the 16 CPU cores was also executing either the dispatcher or another kernel. An overall speedup of 2.3 was achieved over a CPU-only implementation.

The CPU already employs rank reduction. Implementing it on the GPU, could further speed up the GPU computation.

References

[1] "Cray's Titan Supercomputer for ORNL Could Be World's Fastest," http://http://www.pcmag.com/article2/0,2817, 2394515,00.asp.

[2] "Top 500 Supercomputers," http://www.top500.org/ observed May 2012.

[3] "Top 500 Green Supercomputers," http://www.green500.org/lists/2011/11/top/list.php/ observed May 2012.

[4] G. Fann, G. Beylkin, R. J. Harrison, and K. E. Jordan, "Singular operators in multiwavelet bases," *IBM Journal of Research and Development*, vol. 48, no. 2, pp. 161 –171, March 2004.

[5] R. J. Harrison, G. I. Fann, T. Yanai, and G. Beylkin, "Multiresolution quantum chemistry in multiwavelet bases," in *Proc. of the 2003 Intl. Conf. on Comp. Science*, ser. ICCS'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 103–110.

[6] R. J. Harrison, G. I. Fann, T. Yanai, Z. Gan, and G. Beylkin, "Multiresolution Quantum Chemistry: Basic Theory and Initial Applications," *Journal of Chemical Physics*, vol. 121, pp. 11 587–11 598, 2004.

[7] R. J. Harrison and G. I. Fann, "MADNESS: Multiresolution Adaptive Numerical Environment for Scientific Simulation." [Online]. Available: http://www.csm.ornl.gov/ccsg/html/projects/madness.html

[8] R. Harrison and G. Fann, "Speed and precision in quantum chemistry," *SciDAC Review*, 2007.

[9] "BigDFT," http://inac.cea.fr/L_Sim/BigDFT.

[10] NVIDIA, "New GPU applications accelerate search for more effective medicines and higher quality materials," http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09\&version=live\&prid=820175\&releasejsp=release_157\&xhtml=true.

[11] S. Xiao and W. Feng, "Inter-block GPU communication via fast barrier synchronization," 2009.

[12] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel architectures and compilation techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 260–269.

[13] B. Catanzaro, N. Sundaram, and K. Keutzer, "A map reduce framework for programming graphics processors," in *In Workshop on Software Tools for MultiCore Systems*, 2008.

[14] M. Elteir, H. Lin, W. chun Feng, and T. Scogland, "StreamMR: An optimized MapReduce framework for AMD GPUs," in *17th Intl. IEEE Conf. on Par. and Dist. Systems (ICPADS)*, 2011, pp. 364 –371.

[15] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "MapCG: writing parallel program portable between CPU and GPU," in *Proc. 19th Intl. Conf. on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 217–226.

[16] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU clusters," in *Proc. of 2011 IEEE Intl. Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1068–1079.

[17] M. Grossman, A. S. Sbîrlea, Z. Budimlić, and V. Sarkar, "CnC-CUDA: declarative programming for GPUs," in *Proc. 23rd Intl. Conf. on Languages and Compilers for Par. Comp.*, ser. LCPC'10. Springer-Verlag, 2011, pp. 230–245.

[18] K. Knobe, "Ease of use with concurrent collections (CnC)," in *Proc. 1st USENIX Conf. on Hot topics in parallelism*, ser. HotPar'09. USENIX Association, 2009, pp. 17–17.

[19] U. Dastgeer, C. Kessler, and S. Thibault, "Flexible runtime support for efficient skeleton programming on hybrid systems," in *International conference on Parallel Computing (ParCo)*, Gent, Belgium, Aug. 2011.

[20] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, pp. 187–198, 2011.

[21] G. F. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *Proc. 17th Intl. Symp. on High Perf. Dist. Comp.*, ser. HPDC '08. New York, NY, USA: ACM, 2008, pp. 197–200.

[22] R. Dolbeau, S. Bihan, and c. Bodin, Fran "HMPP: A Hybrid Multi-core Parallel Programming Environment," http://www.caps-entreprise.com/upload/ckfinder/userfiles/files/caps-hmpp-gpgpu-Boston-Workshop-Oct-2007.pdf, CAPS Enterprise, Tech. Rep., 2007.

[23] "OpenACC," http://www.openacc-standard.org/.

[24] E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. M. Perez, and E. S. Quintana-Ortí, "A proposal to extend the OpenMP tasking model for heterogeneous architectures," in *Proc. of the 5th Intl. Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, ser. IWOMP '09. Springer-Verlag, 2009, pp. 154–167.

[25] "OpenMP," http://openmp.org/wp.