# High Performance FFT Based Poisson Solver on a CPU-GPU Heterogeneous Platform

Jing Wu
*Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies*
*University of Maryland*
*College Park, MD*
*Email: jingwu@umiacs.umd.edu*

Joseph JaJa
*Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies*
*University of Maryland*
*College Park, MD*
*Email: joseph@umiacs.umd.edu*

***Abstract*—We develop an optimized FFT based Poisson solver on a CPU-GPU heterogeneous platform for the case when the input is too large to fit on the GPU global memory. The solver involves memory bound computations such as 3D FFT in which the large 3D data may have to be transferred over the PCIe bus several times during the computation. We develop a new strategy to decompose and allocate the computation between the GPU and the CPU such that the 3D data is transferred only once to the device memory, and the executions of the GPU kernels are almost completely overlapped with the PCI data transfer. We were able to achieve significantly better performance than what has been reported in previous related work, including over 50 GFLOPS for the three periodic boundary conditions, and over 40 GFLOPS for the two periodic, one Neumann boundary conditions. The PCIe bus bandwidth achieved is over 5GB/s, which is close to the best possible on our platform. For all the cases tested, the single 3D PCIe transfer time, which constitutes a lower bound on what is possible on our platform, takes almost 70% of the total execution time of the Poisson solver.**

***Keywords*-Fast Fourier Transforms; Parallel and Vector Implementations; GPU; CUDA; Poisson Equations**

## I. Introduction

There has been recent interest in the development of high performance direct Poisson solvers due partly to the introduction of immersed-boundary methods [12]. Poisson solvers are an extremely important tool used in many applications, which most often constitute the most computationally demanding component of the application. In an earlier work [18], we developed an FFT-based direct Poisson solver for GPUs, which was optimized for the case when the 3D grid fits onto the device memory. The performance reported there assumes that both the input and output reside on the device memory, which is the typical assumption made by most of the published GPU algorithms. In this work, we consider the case when the grid is much larger than the size of the device memory, but can still fit in the main memory of a multicore CPU, and develop an optimized FFT-based direct Poisson solver on such a platform. Our approach exploits the particular strengths of each processor while carefully managing the data transfers needed between the CPU and the GPU. In particular, our algorithm includes optimized 2D or 3D FFT implementations and optimized tridiagonal solver implementations for such a heterogeneous environment in which both the input and the output reside in the main memory of the CPU.

Most of the recently published work of FFT algorithms on GPUs [8][10][14][13][9][3], assume data sizes limited by the device memory size. This assumption results in efforts that are concentrated on GPU optimization, including data transfers between device memory and the shared memory or registers of the streaming multiprocessors. For memory bound computations, such as FFTs, the performance bottleneck becomes the device memory bandwidth and the type of the global memory accesses. For recent GPUs, the peak device memory bandwidth is higher than 100 GB/s.

We compare our results to two recent results on a similar model. Chen et al [1] used a cluster of 4 or 16 nodes, each node includes two GPUs (Tesla C1060 and GTX 285), to handle large 3D FFT computations. They reported a performance of around 50 GFLOPS on four nodes, somewhat lower than our performance on a single node with a Tesla C1060 (in fact, our performance number is an underestimate since it does not include all the components of our Poisson solver). Another recent work is reported by Gu et al [11], which tries to optimize both CPU-GPU data transfer and GPU computations for 1D, 2D, and 3D FFTs. In particular, they develop a blocked buffered technique for 1D FFTs which achieves a high bandwidth on the CPU-GPU data channel. For their multidimensional FFTs, the data has to be transferred back and forth between the CPU and GPU at least twice, and for 3D double-precision FFT, their best performance is around 15 GFLOPS on the NVIDIA Tesla C2070, 13 GFLOPS on the NVIDIA GTX480 and 9 GFLOPS on the NVIDIA Tesla C1060 respectively. Our performance numbers for the single-precision FFTs reach 60 GFLOPS using the Tesla C1060.

We develop a new approach that introduces the following contributions:

- The computation is organized in such a way that the

3D grid data is transferred between the CPU memory and the device memory only once, while achieving a PCIe bus bandwidth close to the best possible on our platform.
- The GPU kernel computations are almost completely overlapped with the data transfers on the PCIe bus, and hence the GPU execution time contributes very little to the overall execution time. This is due to an effective use of the CUDA page-locked host memory allocation, asynchronous function calls, and write-combining.
- Our approach makes effective use of the multithreaded architectures of both the CPU and GPU. In particular, the FFTs along the X-dimension are partially computed on the multicore CPU in a way that exploits the presence of the processor cores and the L3 cache, and the rest of the computations are carried out on the GPU using a minimum number of coalesced memory accesses with all operations executed directly on the registers.
- While our implementation achieves an accuracy comparable to a double precision implementation, our FFT based direct Poisson solver uses primarily single precision floating operations, which effectively doubles the effective bandwidth of data movement for the same data sizes.
- Experimental tests on our platform for problems of large sizes show that almost $70\%$ of the total execution time is consumed by the single 3D grid data transfer over the PCIe bus, and most of the rest is consumed by the initial CPU computation of the partial FFT along the X dimension. The overall performance of our FFT-based Poisson solver ranges between 50 GFLOPS and 60 GFLOPS.

## II. Overview and Background

In this section, we provide an overview of the algorithms behind the FFT-based Poisson solver, which include FFT and tridiagonal linear system computations. Basic FFT algorithms that are related to our work are then summarized, followed by an overview of Thomas' algorithm for solving tridiagonal linear systems. We end this section with an overview of the general architecture of our platform that consists of a multicore processor with a GPU accelerator.

### A. FFT-based Poisson Solver

The three-dimensional Poisson equation is defined by:

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} = f, \text{ in } \Omega, \tag{1}$$

In our earlier work [17], we presented algorithms for the FFT-based Poisson solver, which were optimized for grid sizes that fit in the device memory. Please refer to [17] for the detailed mathematical formulation and related algorithms. Here, we provide the computational procedures corresponding to a grid of size $I \times J \times K$.

In a nutshell, for the 3 periodic boundary conditions (BC) case, the overall algorithm consists of the following steps:

- Compute the 3D Fast Fourier Transform of the 3 dimensional source dataset $\tilde{f}_{i,j,k}$ to generate $\hat{f}_{l,m,n}$ .
- Divide each $\hat{f}_{l,m,n}$ by a scalar $D_{l,m,n}$ to get the 3 dimensional unknown dataset $\hat{\phi}_{l,m,n}$ .
- Compute the 3D Fast Inverse Fourier Transform of the new 3 dimensional unknown dataset $\tilde{\phi}_{i,j,k}$ to obtain the solution.

We refer to $D_{l,m,n}$ as scalars and to $D_l$, $D_m$ as subscalars defined by:

$$\begin{aligned} D_{l,m,n} &= D_l + D_m + D_n, \ \ where \\ D_l &= 2I^2\left[\cos(2\pi\frac{l}{I}) - 1\right] \\ D_m &= 2J^2\left[\cos(2\pi\frac{m}{J}) - 1\right] \\ D_n &= 2K^2\left[\cos(2\pi\frac{n}{K}) - 1\right] \end{aligned}$$

For the 2 periodic, 1 Neumann BC case, the overall procedure can be described as follows:

- For each value of $k$, $0 \le k \le K-1$, compute the 2D forward Fast Fourier Transform on the corresponding slice of the 3 dimensional source dataset $\tilde{f}_{i,j,k}$ to get $\hat{f}_{l,m,k}$.
- Solve the $I \times J$ tridiagonal linear systems (with size $K \times K$ coefficient matrices) to get $\hat{\phi}_{l,m,k}$.
- For each value of $k$, compute the 2D inverse Fast Fourier Transform on the corresponding slice of the 3 dimensional unknown dataset $\tilde{\phi}_{i,j,k}$.

Clearly both procedures require FFT computations discussed next.

### B. Fast Fourier Transform

The one-dimensional discrete Fourier transform of $n$ complex numbers of a vector X is the complex vector Y defined by:

$$Y[k] = \sum_{j=0}^{n-1} X[j]\omega_n^{jk} \ , \tag{2}$$

where $\omega_n$ is the $n$th root of unity. A fundamental decomposition strategy introduced by the Cooley-Tukey algorithm [2] can be explained through the following equation, where $n = n_1 n_2$.

$$Y[k_1+k_2n_1] = \sum_{j_2=0}^{n_2-1}\left[\left(\sum_{j_1=0}^{n_1-1} X[j_1n_2+j_2]\omega_{n_1}^{j_1k_1}\right)\omega_n^{j_2k_1}\right]\omega_{n_2}^{j_2k_2} \ , \tag{3}$$

Eq (3) expresses the DFT computation as a sequence of three steps. The first step consists of $n_2$ DFT's each of size $n_1$, called radix-$n_1$ DFT, and the second step consists of a set of twiddle factor multiplications (multiplications by $\omega_n^{j_2k_1}$). Finally, the third step consists of $n_1$ DFTs each of size $n_2$, called radix-$n_2$ DFT.

The Cooley-Tukey algorithm can be implemented in a number of ways depending on the recursive structure and the input/output order. Two important variations based on the

recursive structure are the so-called the *decimation in time (DIT)* and the *decimation in frequency (DIF)* algorithms. The *DIT* algorithm uses $n_2$ as the initial radix, and recursively decomposes the DFTs of size $n_1$, while the *DIF* algorithm uses $n_1$ as the initial radix, and recursively decomposes the DFTs of size $n_2$.

Another possible variation of the Cooley-Tukey algorithm stems from the input/output element ordering. For the forward FFT computation, suppose the input is in the original order, the output can either be in bit-reversed order, or in-order; vice versa for the inverse FFT [16].

The advantage of the in-order algorithm [16] is obvious: the output appears in the natural order, which is a key feature of the CUDA FFT library. However, when a DFT or Inverse DFT is used in intermediate steps of a computation, the bit-reverse ordering may provide additional optimization opportunities. In particular, a key feature of the bit-reversed algorithm is that it is an in-place algorithm that overwrites its input with its output data using only O(1) auxiliary storage. The benefits of the in-place algorithm are: 1) the memory requirement is half of the out of place algorithm (potentially doubling the solvable problem size), 2) the butterfly diagrams of the bit-reversed DIF and DIT algorithms are symmetrical [16], which not only indicates symmetrical computation sub-steps, but also a symmetrical memory access pattern. Hence, on the one hand, for GPU computations, the intermediate results for a large size transform can stay in the faster but smaller shared memory and/or registers without being swapped out into the global memory for multiple computation sub-steps. The second feature is of special importance for larger size problems since the in-place algorithm makes it possible to carry out the implementation with a single PCIe back and forth transfer of the grid data.

The multi-dimensional DFT can be defined recursively as a set of DFTs applied to all the vectors along each of the dimensions of a multi-dimensional array. In our algorithms, we use the DIF in order input Cooley-Tukey algorithm on the forward FFT along each dimension and use the DIT in-order output variation on the inverse FFT. We specify corresponding decompositions along each dimension with forward and inverse flag which would be discussed later.

### C. Tridiagonal Linear Systems

Another important component of our Poisson Solver is a tridiagonal linear solver. A tridiagonal solver handles a system of $n$ linear equations of the form $Ax = d$, where $A$ is a tridiagonal matrix, $x$ and $d$ are vectors. This can be represented in matrix form by:

$$\begin{bmatrix} b_0 & c_0 & & & 0 \\ a_1 & b_1 & c_1 & & \\ & a_2 & b_2 & \ddots & \\ & & \ddots & \ddots & c_{n-2} \\ 0 & & & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{n-1} \end{bmatrix}$$

A simplified form of Gaussian elimination, called Thomas' algorithm, is a well-known classical algorithm to solve this problem. The algorithm consists of two sweeps: forward elimination and backward substitution. The forward sweep updates both the vectors $b$ and $d$, and the backward substitution determines the unknown vector $x$.

```
for (int i = 1; i < n; i++) {
      double m = a[i]/b[i-1];
      b[i] = b[i] - m*c[i-1];
      d[i] = d[i] - m*d[i-1];
}
```

**Listing 1:** Forward Elimination

```
x[n-1] = d[n-1]/b[n-1];
for (int i = n - 2; i >= 0; i--)
      x[i]=(d[i]-c[i]*x[i+1])/b[i];
```

**Listing 2:** Backward Substitution

We make the following observations regarding Thomas' algorithm.

- The complexity of the algorithm is $O(n)$, and the algorithm as described seems to be inherently sequential.
- Four one-dimensional arrays for the input $a$, $b$, $c$ and $d$ are needed in the general case.
- It may appear that we need an array for the output $x$ vector; however, the unknown vector can be stored in the $d$ vector during the backward substitution step.

### D. Architecture Overview

Our experimental platform is a heterogeneous processor consisting of a CPU and a GPU. Our CPU consists of two Quad-Core Intel Xeon X5560 with 24 GB of main memory and each quad core shares an 8 MB L3 cache. The GPU is the NVIDIA Tesla C1060 with 4 GB off-chip device memory. Data transfers between the CPU main memory and the GPU device memory are carried by a PCIe 2.0 bus.

*1) CUDA Programming Model:* The CUDA programming model assumes a system consisting of a host CPU and a massively data parallel GPU acting as a co-processor, each with its own separate memory [15]. The GPU executes data parallel functions called kernels using thousands of threads. Kernels can only operate out of device memory.

The basic architecture of our GPU (Tesla C1060 in our experiments) consists of a set of Streaming Multiprocessors(SMs), each of which containing eight Streaming Processors (SPs or cores) executing in a SIMD fashion, 16384 registers, and a 16KB of shared memory. All the SMs have access to the high bandwidth Device memory (peak bandwidth 102 GB/s); such a bandwidth can be exploited only when simultaneous accesses are coalesced into contiguous 16-word lines, but the latency is quite high (around 400-600 cycles).

*2) PCIe bus:* The PCIe 2.0 bus between the CPU and GPU is of central importance for large size problems and for memory bound applications. The PCIe 2.0 has a theoretical peak bandwidth of 8 GB/s, but according to the tests conducted on our Tesla C1060, the best averaged host to device and device to host bandwidth achieved is around 5.7 GB/s. Similar PCIe bus bandwidth was observed in practice by [1] as well. Clearly data transfer between the host and the device memories constitutes the major bottleneck for our problem.

*3) Multi-threaded CPU:* In addition to acting as the CUDA host, the CPU offers a multithreaded environment with a shared memory programming model. In most previous work, the focus is on GPU optimization without trying to make use of the CPU computational resources. In our approach, we make use of the multicore CPU in two ways: 1) we allocated part of the computation to the different cores of the CPU so as to dramatically reduce the data transfer over the PCIe bus; 2) we use the multi-threaded CPU to enable fast transfers between the host memory and the pinned memory.

*4) Asynchronous CUDA streams:* CUDA provides the facility of asynchronous concurrent execution between host and device, for some function calls, control is returned to the host thread before the device has completed the requested task [15]. Data transfer and kernel execution from different CUDA streams [15] can be overlapped when memory copies are performed between page-locked host memory and device memory.

## III. Overall Approach

In this section, we describe our overall strategy to handle the FFT-based direct Poisson solver computations for the cases of three periodic boundary conditions, and the two periodic, one Neumann boundary conditions. In each case, we describe how the overall computation is decomposed and scheduled onto the CPU and GPU and how data transfers between the CPU memory and the GPU global memory are carried out.

### A. Three Periodic BC case

The 3 periodic BC case involves a 3D forward FFT, a scaling of each element, and a 3D inverse FFT. The scaling (division) of each element during the intermediate step depends only on the 3D indices of the element, which allows us to incorporate the scaling operations within the forward FFT or inverse FFT computations. In our implementation, we choose the in-order input FFT DIF variation for the forward FFT, and the in-order output FFT DIT variation for the inverse FFT computation. A straightforward implementation of the 3D FFT algorithm would require bringing the 3D data into the device memory to perform the FFT along each dimension, and hence the 3D data will be passed three times between the CPU and device memories.

We start by noting that the CPU cores offer opportunities for a limited amount of parallelism on highly irregular computations, and that the availability of caches makes the CPU quite effective in handling small size FFTs along the X dimension. On the other hand, the GPU architecture is much more effective for massive data parallel computations with more structured memory access patterns.

Therefore, on the one hand, we try to minimize the number of data movements using the PCIe bus by optimizing the work allocation between the CPU and the GPU; on the other hand, we carefully orchestrate the data movements between the CPU host memory and the GPU global memory to overlap data transfer and kernel execution. In addition, we use a minimum number of fast GPU kernels, using only coalesced global memory accesses with a large number of threads executing the operations directly on the registers.

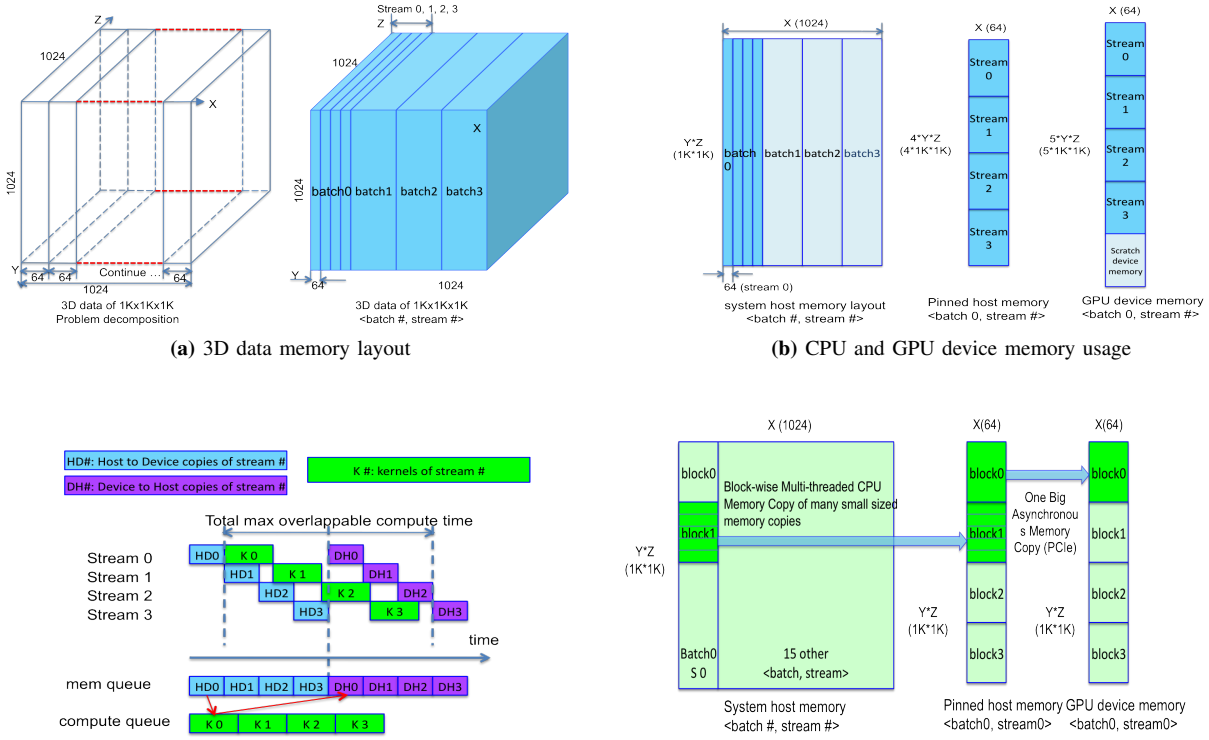
In a nutshell, our solver consists of

- An initial stage of forward small-radix FFT computation along the X dimension using the available CPU cores in an optimal fashion.
- Several batches of multiple asynchronous streams, each involving its 3D data sub-chunk to be transferred from the host system memory to the host pinned memory, then to the device memory, a number of FFT-type GPU kernel functions to be executed on the data, and transfer back the intermediate data into the host system memory eventually.
- The final stage consists of the inverse small-radix FFT computation along the X dimension using the available CPU cores.

We illustrate our strategy in details by focusing on the problem of size $1024 \times 1024 \times 1024$. Similar strategies work as effectively for other sizes.

*1) Multi-threaded CPU forward radix FFT computation:* During the first stage of the forward FFT on the CPU, a set of 64 FFTs, each of radix 16, are computed on the 1024 elements along each dimension X with a stride of 64 between consecutive elements for the same radix FFT. We borrow the following notation from our earlier work [18]: $\{X(64, 16, 64, 1024, forward, CPU)\}$, which amounts to the execution of the first FFT stage on each set of 1024 elements along the X dimension. More precisely, $X(p, q, r, n, forward, CPU)$ indicates that $p$ radix-$q$ forward FFTs along the X dimension are computed over $n$ elements with a stride of $r$ using CPU. Once this step is done, the rest of forward 3D FFT computation is merely a set of 16 independent forward FFTs on data chunks of size $64 \times 1024 \times 1024$. Moreover, the same data chunks of size $64 \times 1024 \times 1024$ can be used to compute the inverse FFT, except for the last stage along the X dimension. The corresponding memory layout of the problem decomposition is shown in Figure 1a.

In order to exploit data locality and maintain independent work for the different threads, we use different threads to




**(a)** 3D data memory layout

**(b)** CPU and GPU device memory usage

**(c)** Async CUDA streams applicable to Tesla C1060

**(d)** block memory copy for one stream

**Figure 1:** Data decomposition and CPU+GPU memory mapping

work on multiple contiguous rows of 1024 elements: each thread in our implementation is responsible for carrying out the 64 sets of FFT(16) for a 1024-element row. Upon completing the work on 8 rows, the 8 threads move onto to the next 8 consecutive rows, and so on. Since there is no dependency between threads, no synchronization is necessary for correctness and such balanced yet independent workload distribution makes a very effective use of the cache by each thread. As a result, we achieve an almost perfect speed-up by a factor of 8 relative to using just one thread.

*2) Asynchronous Streams of Data Transfers and GPU Kernels:* CUDA allows the use of streams for asynchronous memory copy and concurrent kernel executions to hide long PCIe bus latency[15]. A stream is a sequence of commands that execute in order; different streams may execute their commands out of order with respect to one another or concurrently. Asynchronous memory copy has to be carried out between page-locked host memory and device memory.

In order to make effective use of the asynchronous CPU-GPU memory copy, we organize the remaining FFT computations into 4 batches, each consisting of 4 asynchronous streams where each stream involves a subarray of size $64 \times 1024 \times 1024$ (0.5 GB). For our running example, staging page-locked host memory of size 2GB (0.5GB*4) is allocated to enable asynchronous memory copy, as indicated in Figure 1b. By default, page-locked host memory is allocated as cacheable and *write-combining* flag can be used to enable the memory not being snooped during the transfer across the PCIe bus, which can boost the host to device bandwidth in practice[15]. However, the bandwidth on the opposite transfer direction is prohibitively slow. So we allocate two scratch page-locked memory: one with default flag and using for device to host transfer and one with *write-combining* flag and using for host to device transfer. Figure 2 shows one batch of the complete pipelined execution of multi-threaded CPU (including the main thread and the helper threads) and 4 GPU streams (stream 0, 1, 2, 3). Each stream is defined as follows.

- The 3D data subset allocated to each stream is $64 \times 1024 \times 1024$ in the X, Y and Z dimension respectively. This corresponds to the system host memory layout versus <batch #, stream#>in Figure 1d, which indicates $1K \times 1K$ lines of 64 8-byte words with 1024 8-bytes between every two lines. These apart elements need to be packed consecutively in the page-locked memory so that the following PCIe bus transfer would be effective. The data movement for each data subset is a pipeline of block-wise movement involving a multi-threaded

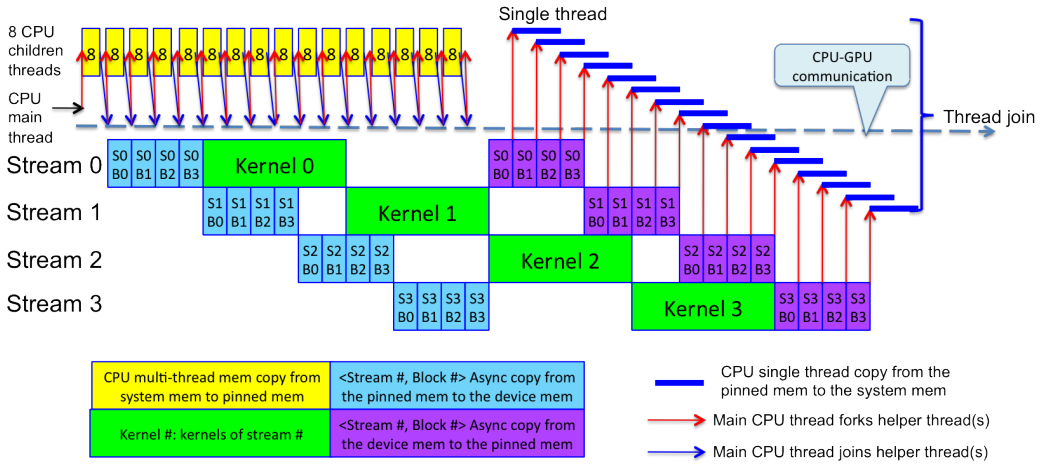



**Figure 2:** CPU-GPU Pipeline

CPU memory copy of a large number of 64-element words into a consecutive block in the paged-locked memory, followed by a PCIe bus transfer. The data movement from the system host memory to the pinned host memory and the data movement from the pinned host memory to the device memory is simultaneous as indicated by the two arrows in Figure 1d. The entire process overlaps PCIe bus transfers with multi-threaded CPU data copy into pinned memory. Due to bandwidth differences of the PCIe bus and the multi-threaded system memory copy, by the time PCIe bus is done with the previous sub-chunk, the next sub-chunk will be ready for the asynchronous memory copy into the device memory. Immediately after we finish the memory copy for one chunk of $64 \times 1024 \times 1024$ data, we launch the asynchronous kernel calls for that stream and start the same work of the next chunk of $64 \times 1024 \times 1024$ data. Upon the completion of the kernel calls, we make use of asynchronous copy attached to the same stream for the copy back. However, due to the limitation of Tesla C1060, there are no concurrent data transfers back and forth between pinned memory and device memory. When we schedule the asynchronous work, we have to schedule the copy back calls after executing all the copying from the pinned host memory to the device memory and their kernel calls. This asynchronous stream execution is shown in Figure 1c.

- Compute the 3D forward FFT, scaling and 3D inverse FFT computation (except for a partial inverse small radix FFT along the X dimension) on a chunk of size $64 \times 1024 \times 1024$ on the GPU using 7 optimized kernels. The total execution time of the kernels should be smaller than the total transfer time of 3 streams (3 host to device and 3 device to host, Figure 1c); otherwise, one or more of the streams' memory transfer back needs to be held back until the completion of its kernel. This is illustrated in Figure 1c. Since we want to achieve a high PCIe bus bandwidth, the kernels have to execute as fast as well. Once the data is loaded onto the GPU device memory, we can use techniques similar to those introduced in our previous work [18] to compute the FFT of each subarray of size $64 \times 1024 \times 1024$. In particular, the X-dimensional radix-64 forward FFT and inverse FFT are included in the Y and Z dimensional FFT computation kernels to avoid additional global memory accesses. Moreover, effective shared memory transpose strategies are used to guarantee no bank conflicts. An intermediate global memory (shared by 4 streams due to their sequential execution of kernels) is introduced for smaller strides between consecutive global memory accesses when multiple Z dimensional computation kernels are involved (Figure 1b), without limiting the maximum number of concurrent streams. The scaling step is included in the last step of the forward FFT and the first step of the inverse FFT with the scalars computed using bit-reversed indices.
  As a result, the GPU kernels can be defined by the following computations:
  - $\{X(8,8,8,64), Y(32,32,32,1024), forward\}$
  - $\{X(8,8,1,8), Y(32,32,1,32), forward)\}$
  - $\{Z(32,32,32,1024), forward\}$
  - $\{Z(32,32,1,32), forward\}$
    $\{scaling, GPU)\}$
    $\{Z(32,32,1,32), inverse\}$
  - $\{Z(32,32,32,1024), inverse\}$
  - $\{X(8,8,1,8), Y(32,32,1,32), inverse\}$
  - $\{X(8,8,8,64), Y(32,32,32,1024), inverse\}$

  Note that all the arithmetic computations are carried out on register contents, all global memory transfers involve coalesced memory access, and transpose computations use shared memory without any bank conflicts. Therefore, we complete the $64 \times 1024 \times 1024$ forward FFT, scaling and inverse FFT using 7 kernels.

- Once the kernels are completed, we perform block-wise asynchronous memory copies from the device memory to the pinned host memory and then to the system host memory for each stream. CPU function calls are synchronous in nature, and CPU memory copy back calls have to wait for the completion of the asynchronous GPU-to-CPU memory copy for that data chunk (Figure 2).

*3) Multi-threaded CPU inverse radix FFT computation:* This step is similar to the very first step to compute partial FFTs along the X dimension using the 8 cores of the CPU. Such a computation can be described by $\{X(64, 16, 64, 1024, inverse, CPU)\}$

## IV. 2 Periodic 1 Neumann Poisson Solver

### A. Algorithm

Suppose our 3D input data is of size IxJxK. The 2 periodic 1 Neumann BC case involves K sets of 2D forward FFTs, each of size IxJ, followed by IxJ sets of tridiagonal linear systems, each of size KxK, followed by K sets of 2D inverse FFT of size IxJ. We have a similar computation decomposition and distribution on the CPU and GPU, as the three periodic BC case.

### B. Strategy

We illustrate our strategy for the case of $1024 \times 1024 \times 1024$ , and examine in some detail how the work is allocated between the CPU and the GPU. The same strategy works for other problem sizes as we demonstrate later. The overall approach can be described as follows.

- As before, the first step is carried out on the CPU, and partially computes the forward FFT along the X dimension using the scheme described by: $\{X(64, 16, 64, 1024, forward, CPU)\}$, i.e., 64 FFTs each of radix-16, stride-64, for each 1024-element row.
- Launch a set of asynchronous streams involving memory copy and each of the streams performs the following computations of data size $64 \times 1024 \times 1024$ running on the GPU:
  - Compute the forward radix-64 FFT along the dimension X as described by: $\{X(1, 64, 1, 64, forward, GPU)\}$ and the forward FFTs along the Y dimension: $\{Y(1, 1024, 1, 1024, forward, GPU)\}$
  - Using Thomas' algorithm, solve the tridiagonal linear systems of equations $\{Z(1024, tridiagonal\ solver, GPU)\}$
  - Compute the inverse FFT along the Y dimension: $\{Y(1, 1024, 1, 1024, inverse, GPU)\}$ and the inverse radix-64 FFT along the X dimension: $\{X(1, 64, 1, 64, inverse, GPU)\}$
- After the GPU completes the execution of all the kernels and the intermediate results are written back in the CPU main memory, we execute the following inverse FFT computation: $\{X(64, 16, 64, 1024, inverse, CPU)\}$ using the 8 CPU cores.

Note that, once a chunk is loaded into the GPU global memory, we ensure a fast GPU execution by minimizing the number of global memory accesses, all of which are guaranteed to be coalesced. Similar as in the 3D periodic case, the X dimensional radix-64 forward and inverse FFTs can be included in the 2 kernels within the Y dimensional size 1024 forward and inverse FFTs. Such a GPU allocation allows us to use 64 CUDA threads to process the Y and/or Z dimensional computations in a vector-wise manner, which naturally guarantee coalesced global memory access of all data.

CUDA streams are employed to combine the CPU and GPU work using asynchronous memory copy and kernel executions in a similar way to what we did for the 3 periodic BC case: 4 streams achieve typically a very good PCIe bandwidth (around 5GB/s) in our experiments.

### C. Arithmetic Precision

When it comes to GPU performance, single precision floating point arithmetic enjoys significant benefits over double precision arithmetic[7]. Since single precision floating points use half of the memory space of double precision floating points, single precision implementations potentially save half of the memory transfer time, for the PCIe bus and for the global memory accesses. Also, single precision computations are faster than double precision computations in many architectures, including the Tesla C1060 we are using. An important characteristic of our algorithm is to secure a $2^{nd}$ order convergence, and hence if we make the grid twice as dense, we expect the accuracy to double. In our experiments, double precision arithmetics can easily guarantee such property at the expense of slower computation time, while pure single precision implementations showed a relatively larger error when compared to the discretized analytic function used in our tests. And due to the slow PCI peak bandwidth and fast GPU kernels, these two variations show almost the same performance in our experiments.

To achieve high performance while ensuring the $2^{nd}$ order convergence, we make use of a precision boost for the intermediate data. Through careful examination, we notice that the step that most affects the precision is the division step in the forward elimination stage: $m = a[i]/b[i-1]$. More specifically, the error becomes large when $b[i-1]$ is small. Note that in our implementation, the $b[i-1]$ is stored and updated as we iterate along $i$. Hence we use double precision to store the $b[i-1]$ values and immediately related variables, and then cast the results back into single floating points. By using this trick, we can avoid the substantial overhead of converting the entire data into double precision while achieving the desired accuracy.

**Table I:** Basic Parameters of Tesla C1060 (SM: Streaming Multiprocessor; SP: Streaming Processor)

| | # of SMs | # of SPs | # of Registers | Shared Mem. | Global Mem. | Nvcc Cufft |
|---|---|---|---|---|---|---|
| Tesla C1060 | 30 | 8 | 16K | 16KB | 4GB | 3.2.16 |

**Table II:** CPU configuration

| CPU | Freq. | Cores | Sys. Mem. | GCC | FFTW |
|---|---|---|---|---|---|
| Xeon X5560 | 2.80GHz | 2 quad-core | 24 GB | 4.7.2 | 3.3.2 |

## V. Performance

In this section, we present a summary of the performance tests that have been conducted on our CPU-GPU platform. Our CPU consists of two Quad-Core Intel Xeon X5560, each Quad-core with an 8 MB L3 cache, such that the total main memory is of size 24 GB . Our GPU card is an NVIDIA Tesla C1060 GPU, and data transfer between the CPU main memory and the GPU device memory is through PCIe 2.0 bus. Tables I and II provide more detailed information about our GPU and CPU configurations.

In our tests, the problem size is a power-of-two in each of the three dimensions. We use input sizes that cannot be accommodated by the device memory alone (in particular, we would not be able to run CUFFT on the sizes that should in principle fit the device memory).

Since the core of our algorithms is based on either 3D or 2D FFT computations, we use the following well-known formula to estimate the FFT GFLOPS performance, assuming that the execution time of a one-dimensional FFT on data size $NX$ is $t$ seconds:

$$GFLOPS = \frac{5 \cdot NX \cdot \log_2(NX) \cdot 10^{-9}}{t} \quad (4)$$

At some point, we compare the performance of our FFT implementations against implementations obtained by the single-precision multi-threaded Single Instruction Multiple Data (SIMD) code enabled version FFTW library [6][4][5] (2D or 3D). We use "MEASURE" patient level when generating the execution plan since using a more rigorous patient level will take over 8 hours to generate the execution plan for the problem sizes we are dealing with.

### A. The Case of Three Periodic Boundary Conditions

Our three periodic BC Poisson solver consists of a forward 3D FFT, a scaling (division) step for each element of the intermediate 3D array, followed by an inverse 3D FFT. Therefore, the number of GFLOPS achieved by our algorithm can simply be calculated based on the 3D FFT GFLOPS formula. Since we do not include the intermediate scaling step in our estimate, we under-estimate the performance of our algorithm. Specifically, if the total execution time on a 3D data set of size $NX \times NY \times NZ$ is $t$ seconds, then its GFLOPS can be measured using the standard formula:

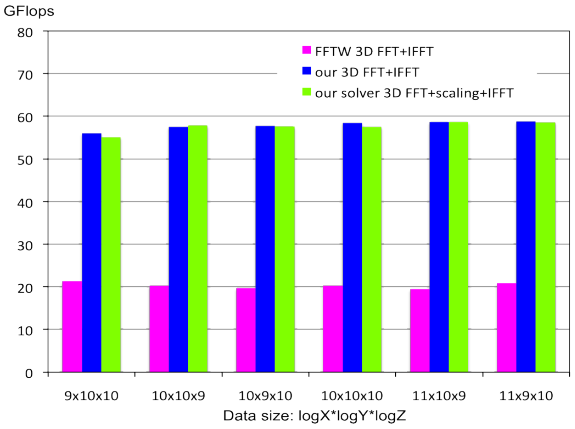
$$GFLOPS = \frac{2 \cdot 5 \cdot NX \cdot NY \cdot NZ \cdot [\log_2(NX \cdot NY \cdot NZ)] \cdot 10^{-9}}{t} \quad (5)$$

The coefficient 2 in the above formula captures the forward and the inverse FFT. Figure 3a illustrates the GFLOPS performance of our Poisson solver and ourcombined 3D forward and inverse FFTs. We have also included the performance of the 3D forward and inverse FFTW running on our CPU as a point of reference. As can be seen from the figure, we were able to achieve more than 55 GFLOPS for all data sizes for our Poisson solver and for our combined forward and inverse FFTs. The best that we were able to achieve using FFTW is around 20 GFLOPS.

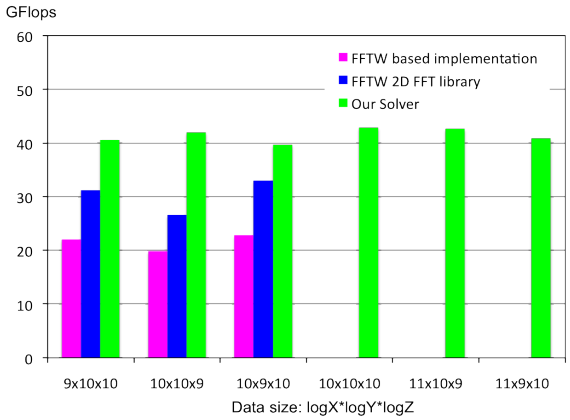
As stated in the introduction, and as illustrated in the above figure, our performance is substantially better than those reported in the most recent related works of [1] and [11].

We now compare the PCIe transfer time to the total Poisson solver time. Figure 3c shows the ratio of the best possible achievable PCIe transfer time and the total solver time for several 3D grid sizes. The best achievable bus transfer time is computed by simply moving the corresponding data from main memory to the GPU global memory, and immediately writing it back to the CPU main memory. As can be seen from this figure, more than 50% of the total solver time is consumed in moving the data once between the CPU main memory and the device memory. The best previous algorithm for the 3D forward FFT moved the data twice, and hence incur a substantial overhead compared to our algorithm, even without including the scaling and the inverse FFT steps. Figure 3c also shows the ratio of the total GPU time of our solver versus the entire solver time. The total GPU time starts from the moment the CPU begins to copy data from the system host memory to the pinned host memory and ends at the moment that all the GPU computations are completed and the output is stored back in the original system host memory. As we can see from the figure for the 3 periodic BC case, the GPU related time (which in fact results from one round of PCIe bus memory transfer) represents around 60%-70% of the total runtime.

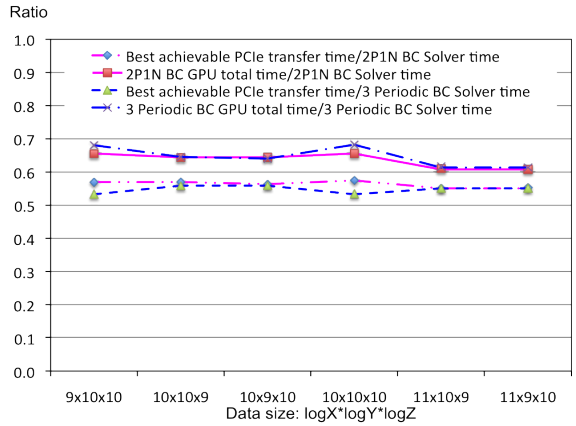
We now turn our attention to our CPU partial FFT computation along the X dimension. The allocation of the this computation to the CPU has enabled us to perform the whole computation with just one transfer of the 3D data over the PCIe bus. Figure 4 demonstrates the scalability of the CPU implementation relative to the number of threads for different sizes. On the other hand, Figure 3c provides the ratio of CPU part radix FFT runtime and the best achievable PCIe bus bandwidth on the same data size. Therefore with 4 or more threads, the execution time of this part of the computation is substantially less than the best achievable transfer time of the 3D data between the CPU main memory and the GPU global memory. In fact, with 8 threads, the



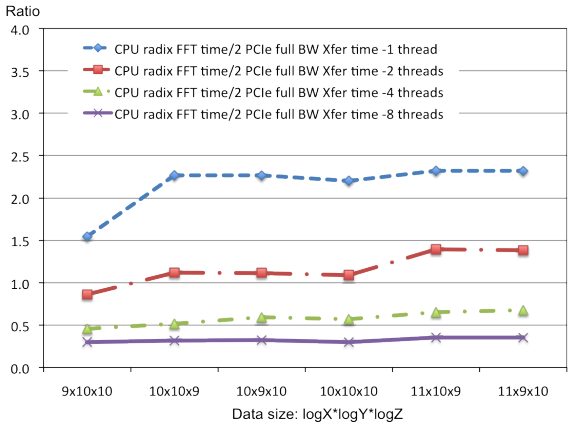

**(a)** 3 Periodic BC Performance Comparison



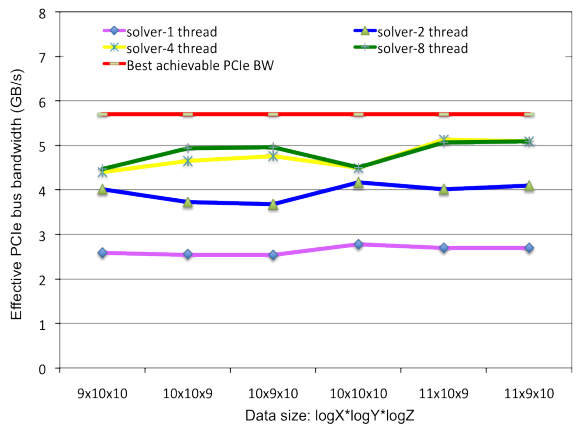

**(b)** 2 Periodic 1 Neumann BC Performance Comparison



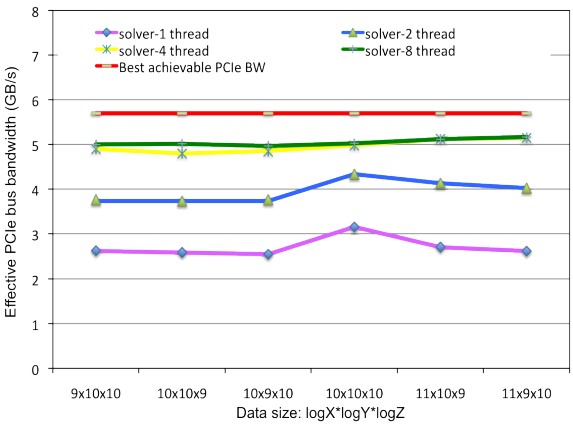

**(c)** Ratios of PCIe transfer time to solver time, GPU solver time, for both types of boundary conditions




**(d)** CPU radix FFT runtime vs. PCI transfer time




**(e)** 3 Periodic BC BW vs # of Threads




**(f)** 2 Periodic 1 Neumann BC BW vs # of Threads

**Figure 3:** Performance Evaluation Summary

CPU radix-FFT time is around 40% of the best possible data transfer time over the PCIe bus.

We now take a closer look at the PCIe bandwidth achieved by our solver and at how we were able to overlap the execution of the GPU kernels with the CPU-GPU asynchronous memory copy enabled by the asynchronous CUDA streams.

We evaluate the effective PCIe bandwidth using Formula

**Table III:** Arithmetic Accuracy of 3D Periodic BC Solver

| Data size | Our Solver | Using FFTW |
|---|---|---|
| $512 \times 1024 \times 1024$ | 0.000028 | 0.000028 |
| $1024 \times 512 \times 1024$ | 0.000012 | 0.000013 |
| $1024 \times 1024 \times 512$ | 0.000028 | 0.000028 |
| $1024 \times 1024 \times 1024$ | 0.000011 | 0.000012 |
| $2048 \times 1024 \times 512$ | 0.000024 | 0.000024 |
| $2048 \times 512 \times 1024$ | 0.000008 | 0.000008 |

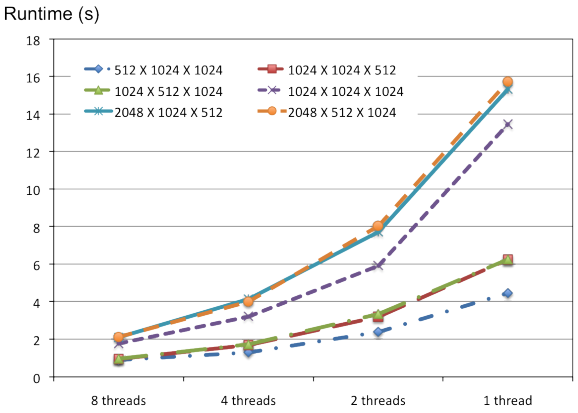



**Figure 4:** CPU radix FFT runtime scalability

6:

$$BW = \frac{2 \cdot 8 \cdot NX \cdot NY \cdot NZ \cdot 10^{-9}}{t} \tag{6}$$

where 8 in the formula is due to the fact that the number of bytes occupied by each single floating point complex element is 8 and 2 indicates moving the data from the CPU to the GPU and then back to the CPU.The time $t$ used in the above formula excludes the CPU runtime for the first stage of X dimensional forward FFT and the last stage of X dimensional inverse FFT. The time $t$ starts from the momentthat the CPU begins to copy data from the system host memory to the pinned memory for the asynchronous memory copy, and ends at the moment that all the complete results are copied back into the system host memory. Figure 3e demonstrates the achieved PCIe bus bandwidth when different numbers of threads are used when copying data from the system host memory to the pinned memory in blocks. We use multi-threading memory copy for each block, followed by an immediate CPU to GPU asynchronous memory copy call for that block.

Finally, Table III shows the maximum absolute difference between the results obtained by our solver and the discretized the analytic function $\cos(2\pi x) \cdot sin(2\pi y) \cdot \cos(2\pi z)$ on the grid used to generate our input data. Our solver demonstrates similar accuracy compared to the FFTW library single precision.

### *B. The Two-Periodic and One-Neumann BC Case*

As described before, for the problem of size $NX \times NY \times NZ$, the two periodic and one Neumann BC case Poisson

**Table IV:** Arithmetic Accuracy of 2 Periodic 1 Neumann BC Solver

| Data size | Our Solver | Using FFTW |
|---|---|---|
| $512 \times 1024 \times 1024$ | 0.000008 | 0.000006 |
| $1024 \times 512 \times 1024$ | 0.000008 | 0.000007 |
| $1024 \times 1024 \times 512$ | 0.000006 | 0.000006 |
| $1024 \times 1024 \times 1024$ | 0.000008 | NA |
| $2048 \times 512 \times 1024$ | 0.000008 | NA |
| $2048 \times 1024 \times 512$ | 0.000005 | NA |

Solver consists of $NZ$ number of 2D forward FFTs, each of size $NX \times NY$, $NX \times NY$ tridiagonal linear systems with matrix size $NZ \times NZ$, and $NZ$ number of 2D inverse FFTs, each of size $NX \times NY$. We conduct a similar experimental tests as those carried out for 3 periodic BC case; however, we employ a GFLOPS formula that is appropriate for the corresponding computations. The number of GFLOPS now consist of two components: the 2D FFT computations, and the 1D tridiagonal solvers.

The 2D FFT or IFFT component can be easily captured as follows. If the execution time of 2D FFT or IFFT on data of size $NX \times NY$ is $t$ seconds, then

$$GFLOPS = \frac{5 \cdot NX \cdot NY \cdot [\log_2 (NX \cdot NY)] \cdot 10^{-9}}{t} \tag{7}$$

The number of GFLOPS needed to solve the tridiagonal linear system of size $N$ is $8N$, and hence the total GFLOPS formula for the 2 periodic (say, the X and Y dimensions) 1 Neumann (say, Z dimension) BC is the following:

$$GFLOPS = \frac{NX \cdot NY \cdot NZ \cdot [10 \cdot \log_2 (NX \cdot NY) + 8] \cdot 10^{-9}}{t} \tag{8}$$

The total GFLOPS performance of our Poisson solver for this case is shown in Figure 3b. As can be seen from this figure, we were able to achieve over 40 GFLOPS for all the problem sizes tested. We now compare our implementation against an implementation that uses a single-precision FFTW library to compute the 2D FFTs and an optimized Thomas algorithm based tridiagonal solver. The tridiagonal solver is customized to the Poisson Solver for better performance and is essentially a double precision solver with single-precision input and output data. Double precision is necessary to achieve second order accuracy. Intermediate memory transpositions were needed between the FFT and the tridiagonal solver to guarantee the efficiency of both the FFTW library and the tridiagonal solver; however, we exclude the transposition time when evaluating the performance to indicate a best possible achievable performance using the standard library. Figure 3b demonstrates the comparison between the two implementations. As we can see, our solver demonstrates superior performance in all cases even if we just compare to the 2D FFTW library to our solver, which includes a tridiagonal solver.

On the other hand, in terms of the effective PCIe bandwidth, we still use of the same Formula 6 since the same type of data movements are occurring as before.

Due to the sufficient number of concurrent CUDA streams, and relatively smaller total time of the GPU kernels of each stream, we were able to better hide the GPU kernel executions which yields a higher effective PCIe transfer bandwidth. As we can observe from Figure 3f, around 5 GB/s effective PCIe bandwidth is achieved for all data sizes by using 4 or 8 CPU threads for memory copy.

Finally, we compare in Table IV the accuracy using our single precision solver on the discretized analytic function $sin(2\pi x + \frac{\pi}{10}) \cdot \cos(2\pi y) \cdot \cos(2\pi z)$ on the grid with the precision boost step against the naive standard method implementation with single precision FFTW and double precision tridiagonal solver. Due to the additional memory used for intermediate array transposition, some larger sized problems could not be completed. As can be seen from the table, our solver achieves similar accuracy as the FFTW single precision algorithm and the double precision tridiagonal solver.

## VI. Conclusion

We presented in this paper a new strategy to map an FFT-based direct Poisson solver on a CPU-GPU heterogeneous platform, which optimizes the problem decomposition using both the CPU and the GPU. The new approach effectively pipelines the PCIe bus transfer and GPU work, almost entirely overlapping the CPU-GPU memory transfer time and the GPU computation time. Experimental results over a wide range of grid sizes have shown very high performance, both in terms of the number of floating point operations per second and the effective PCIe bus memory bandwidth. The performance numbers are superior to those that were achieved in previous related work.

## Acknowledgment

This work was partially supported by an NSF PetaApps award, grant OCI0904920, the NVIDIA Research Excellence Center at the University of Maryland, and by an NSF Research Infrastructure Award, grant number CNS 0403313.

## References

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

[2] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[3] Y. Dotsenko, S. Baghsorkhi, B. Lloyd, and N. Govindaraju. Auto-tuning of fast Fourier transform on graphics processors. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 257–266, New York, NY, USA, 2011. ACM.

[4] M. Frigo. A fast Fourier transform compiler. *SIGPLAN Not.*, 34(5):169–180, May 1999.

[5] M. Frigo and G. Johnson. The FFTW website, 2012. http://www.fftw.org.

[6] M. Frigo, Steven, and G. Johnson. The design and implementation of FFTW3. In *Proceedings of the IEEE*, pages 216–231, 2005.

[7] D. Goddeke and R. Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):22–32, Jan. 2011.

[8] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[9] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[10] L. Gu, X. Li, and J. Siegel. An empirically tuned 2D and 3D FFT library on CUDA GPU. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 305–314, New York, NY, USA, 2010. ACM.

[11] L. Gu, J. Siegel, and X. Li. Using GPUs to compute large out-of-card FFTs. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 255–264, New York, NY, USA, 2011. ACM.

[12] R. Mittal and G. Iaccarino. Immersed boundary methods. In *Ann. Rev. Fluid Mech. 37*, pages 239–261, 2005.

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

[14] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 5:1–5:11, Piscataway, NJ, USA, 2008. IEEE Press.

[15] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2011.

[16] B. C. Sidney. Fast Fourier Transforms. Appendix 1: FFT flowgraphs, 2012. http://cnx.org/content/m16352/latest/?collection=col10550/1.20.

[17] J. Wu and J. JaJa. An optimized FFT-based direct Poisson solver on CUDA GPUs. *IEEE Trans. Parallel Distrib. Syst.* To appear.

[18] J. Wu and J. JaJa. Optimized strategies for mapping three-dimensional FFTs onto CUDA GPUs. In *Innovative Parallel Computing (INPAR)*. IEEE Press, 2012.