ARTICLE IN PRESS

J. Parallel Distrib. Comput. ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Optimized FFT computations on heterogeneous platforms with application to the Poisson equation

Jing Wu *, Joseph JaJa

Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, United States

HIGHLIGHTS

- New strategy to decompose large multi-dimensional FFTs on CPU–GPU platforms.
- Executions of GPU kernels are almost completely overlapped with PCI bus transfer.
- Multi-dimensional data is transferred only once between the GPU and CPU.
- Scheme is equally effective for the single and double precision computations.

ARTICLE INFO

Article history:
Received 21 August 2013
Received in revised form
18 March 2014
Accepted 21 March 2014
Available online xxxx

Keywords:
Fast Fourier transforms
Parallel and vector implementations
CUDA GPU
Poisson equations

ABSTRACT

We develop optimized multi-dimensional FFT implementations on CPU–GPU heterogeneous platforms for the case when the input is too large to fit on the GPU global memory, and use the resulting techniques to develop a fast Poisson solver. The solver involves memory bound computations for 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 145 GFLOPS for the three periodic boundary conditions (single precision version), and over 105 GFLOPS for the two periodic, one Neumann boundary conditions (single precision version). The effective bidirectional PCIe bus bandwidth achieved is 9–10 GB/s, which is close to the best possible on our platform. For all the cases tested, the single 3D data 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.

1. Introduction

There has been recent interest in the development of high performance direct Poisson solvers due partly to the introduction of immersed-boundary methods [13]. A Poisson solver is an extremely important tool used in many applications, which most often constitutes the most computationally demanding component of the application. In an earlier work [19], 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 host multicore CPU, and develop optimized FFT computations, and FFT-based direct Poisson solver on such platforms, which significantly expand our earlier work in [20]. 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 heterogeneous environments 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 [6,8,15,14,7,3], 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

* Corresponding author.
E-mail addresses: jingwu@umiacs.umd.edu (J. Wu), joseph@umiacs.umd.edu (J. JaJa).

http://dx.doi.org/10.1016/j.jpdc.2014.03.009
0743-7315/

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 can be 100–200 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 under-estimate since it does not take into consideration all the components of our Poisson solver). Another recent work is reported by Gu et al. [9], 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 multi-dimensional 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. And when using Tesla K20, which supports bidirectional PCIe bus transfers (similar as Tesla C2070), we achieved more than 140 GFLOPS for the single precision FFTs and more than 70 GFLOPS for the double precision FFTs.

Our main contributions can be summarized as follows.

- 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 platforms.
- 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, stream scheduling, and write-combining.
- Our CPU–GPU workload decomposition is equally effective for both the single precision and double precision implementations. While our single precision implementation achieves an accuracy comparable to a double precision implementation, it achieves double the GFLOPS 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 FFT along the X dimension. The overall performance of our FFT-based Poisson solver is in ranges of 50–60 GFLOPS for a relatively older CPU–GPU platform and around 140 GFLOPS for a newer platform.

2. 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 platforms that consists of a multicore processor with a GPU accelerator.

2.1. FFT-based Poisson solver

The three-dimensional Poisson equation is defined by:

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

In our earlier work [21], we presented algorithms for the FFT-based Poisson solver, which were optimized for grid sizes that fit in the device memory. Please refer to [21] 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:

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

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.

2.2. 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 ω_n is the nth root of unity. A fundamental decomposition strategy introduced by the Cooley–Tukey algorithm [2] can be explained through the following equation, where $n = n_1 n_2$.

$$Y[k_1 + k_2 n_1] = \sum_{j_2=0}^{n_2-1}\left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2]\omega_{n_1}^{j_1 k_1}\right)\omega_n^{j_2 k_1}\right]\omega_{n_2}^{j_2 k_2}. \tag{3}$$

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 $\omega_n^{j_2 k_1}$). 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.

Please cite this article in press as: J. Wu, J. JaJa, Optimized FFT computations on heterogeneous platforms with application to the Poisson equation, J. Parallel Distrib. Comput. (2014), http://dx.doi.org/10.1016/j.jpdc.2014.03.009

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 [18].

The advantage of the in-order algorithm [18] 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 [18], which not only indicate 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.

Many parallel FFT libraries have been designed to exploit the computing power of modern processors: both on general purpose CPUs and on accelerators such as GPUs. For multicore CPUs, widely used FFT libraries include the FFTW [4] library and Intel's MKL [11]. The FFTW is known as the fastest free software that achieves high performance by auto-tuning the decomposition that most suits the target hardware architecture. Intel has their own version of FFT implementation as part of the Intel MKL, which seems to perform extremely well for Intel architecture-based platforms. On the other hand, for GPU implementations, CUFFT [17] typically leads the in-card sized FFT performance on NVIDIA latest GPUs. At the time of this revision, CUDA Version 6.0 just introduced a "multi-GPU" version of CUFFT which is limited to two GPUs on the same board (such as a Tesla K10), in addition to restrictions of the library functionality.

2.3. Tridiagonal linear systems

An 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 inputs 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.

2.4. Architecture overview

Our experimental platforms are heterogeneous processors, each of which consists of a multicore CPU and a GPU accelerator, such that the CPU memory is substantially larger than the GPU device memory. More specifically, we use two testbeds for our work. The first is a dual socket quad-core Intel Xeon X5560 CPU with 24 GB main memory and an NVIDIA Tesla C1060 with 4 GB device memory—we refer to this testbed as the *Nehalem–Tesla* node, after the codename of the CPU and the architecture of the GPU, respectively. The second is a dual socket octal-core Intel E5-2690 with 128 GB main memory and an NVIDIA Tesla K20 with 5 GB device memory—we refer to this testbed as the *Sandy–Kepler* node (we use Sandy rather than Sandy Bridge for brevity). The input data is much larger than the device memory and is assumed to be initially held in the CPU memory. At the end of the computation, the output data must reside in the CPU memory as well. Data transfers between the CPU main memory and the GPU device memory are carried out by a PCIe 2.0 bus: unidirectional for the *Nehalem–Tesla* node (compute capability 1.3) and bidirectional for the *Sandy–Kepler* node (compute capability 3.5).

2.4.1. 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 [16]. The GPUs consist of a number of Streaming Multiprocessors (SMs), each of which contains a number of Streaming Processors (SPs or cores). The GPU executes data parallel functions called kernels using thousands of threads. The mapping of threads onto the GPU cores are abstracted from the programmers through—(1) a hierarchy of thread groups, (2) shared memories, and (3) barrier synchronization. Such abstraction provides fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism and this is based on similar hardware architecture among generations. Details of the CUDA programming model can be found at [16] and we will only refer to the aspects that are key to our optimization scheme. In this work, we are concerned with Tesla C1060 and K20 whose main features are summarized in Table 1. Note that, for the Tesla K20, the L1 cache and the shared memory per SM share a total amount of 64 kB on-chip memory whose ratio is configurable.

Please cite this article in press as: J. Wu, J. JaJa, Optimized FFT computations on heterogeneous platforms with application to the Poisson equation, J. Parallel Distrib. Comput. (2014), http://dx.doi.org/10.1016/j.jpdc.2014.03.009

Table 1
The two GPUs used in this paper.

GPUs	Tesla C1060	Tesla K20
SMs per GPU	30	14
SPs per SM	8	192
Registers per SM	16 K	64 K
Shared mem per SM	16 kB	16–48 kB
L1 cache per SM	NA	48–16 kB
L2 cache per GPU	NA	1.25 MB
Global Mem per GPU	4 GB	5 GB
GPU clock rate	1296 MHz	706 MHz
Memory clock rate	800 MHz	2600 MHz
Memory bandwidth	102.4 GB/s	208 GB/s
Compute capability	1.3	3.5

2.4.2. PCIe bus

The PCIe 2.0 bus between the CPU and GPU is of central importance for large size problems and for memory bound computations such as ours. The PCIe 2.0 has a theoretical single directional peak bandwidth of 8 GB/s—with a relatively smaller best achievable bandwidth in our evaluation. On the *Sandy–Kepler* node, single directional memory transfer from pinned host memory to device memory (H2D-host to device) reaches 5.7 GB/s bandwidth and from device memory to pinned host memory (D2H-device to host) achieves 6.2 GB/s bandwidth. However, when bidirectional memory transfer is done concurrently, a further slight bandwidth degradation is observed: 5.44 GB/s for D2H and 5.34 GB/s for H2D are the best we were able to achieve for pure bidirectional memory transfer with varying data sizes. This gives a combined 10.78 GB/s upper bound on the best achievable bandwidth. On the *Nehalem–Tesla* node, only single directional memory transfer is supported and the observed H2D bandwidth is 5.4 GB/s and for a D2H copy the best bandwidth achievable is 5.3 GB/s. Similar observations were reported by others including NVIDIA [12,9]. Clearly the data transfer between the host and the device memories constitutes the major bottleneck for our problem.

2.4.3. Multicore CPU

In addition to acting as the CUDA host, the multicore CPU offers in itself a multi-threaded environment with a shared memory programming model. In most previous work, the focus has been 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 allocate part of the computation to the CPU cores and partition the CPU and GPU work in such a way that the GPU work requires only one iteration of data transfer over the PCIe bus; (2) we use the multicore CPU to enable concurrent asynchronous transfers between the host memory and the pinned memory: unidirectional for the *Nehalem–Tesla* node and bidirectional for the *Sandy–Kepler* node. In addition, modern multicore CPUs are built with SIMD support: SSE is supported on the Xeon X5560 and AVX is supported on the Xeon E5-2690. Such features allow us to carry out a limited amount of data intensive parallel computations quite effectively on the CPU.

2.4.4. Asynchronous CUDA streams

CUDA supports asynchronous concurrent execution between host and device through some asynchronous function calls—control is returned to the host thread before the device has completed the requested task [16]. Data transfer and kernel execution from different CUDA streams [16] can be overlapped when memory copies are performed between page-locked host memory and device memory. Some devices of compute capability of 2.x and higher (K20 in our evaluation) can perform memory copy from page-locked host memory to device memory (H2D) concurrently with a copy from device memory to page-locked host memory (D2H). With careful orchestration of the CPU work and CUDA streams, we essentially establish a CPU–GPU work pipeline of depth of four (for the *Nehalem–Tesla* node) and five (for the *Sandy–Kepler* node) in which computation and communication are almost completely overlapped. Moreover, our effective CPU–GPU work pipeline of bidirectional PCIe bus transfer essentially doubles the PCIe bus performance of the unidirectional PCIe bus transfer version.

3. Overall approach

In this section and the following 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 each of the CPU–GPU platforms, and how data transfers between the CPU memory and the GPU global memory are managed to cause an almost complete overlap between computation and data transfer.

3.1. Three periodic boundary condition case

The 3 periodic Boundary Condition (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 moving the 3D data once along each dimension, resulting in the 3D data being exchanged between the CPU and the GPU over the PCIe bus three times.

We start by noting that the CPU cores offer opportunities for a significant amount of parallelism on highly irregular computations, and that the availability of caches makes the CPU quite effective in handling FFTs along the X dimension due to the memory layout of the 3D data. Note also that the SIMD capability of the CPU presents possibilities for additional performance enhancement. On the other hand, the GPU architecture is much more effective for massive data parallel computations using more structured memory access patterns. Therefore, we decompose the overall work among the CPU and the GPU in such a way that: (1) the volume of the data transferred over the PCIe bus is minimized. In our case, the 3D data will be transferred only once between the two devices; (2) the FFT computations along the X dimension will be effectively carried out by the CPU cores; and (3) the rest of the FFT computations will be carried out by the GPU cores through a sequence of asynchronous streams of chunks of the 3D data. Each asynchronous stream will go through a 5-stage pipeline consisting of: data transfer from the host system memory to the host pinned memory; memory copy from the host to the device memory (H2D); GPU kernel executions; memory copy from the device to the host pinned memory (D2H); and data transfer to the host system memory. We orchestrate the data movements to overlap H2D memory copy, kernel execution, and D2H memory copy.

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

3.1.1. Multi-threaded CPU forward X dimensional FFT

As mentioned before, the FFT computations along the X dimension are carried out by the CPU cores. We make use of Intel's Math

Please cite this article in press as: J. Wu, J. JaJa, Optimized FFT computations on heterogeneous platforms with application to the Poisson equation, J. Parallel Distrib. Comput. (2014), http://dx.doi.org/10.1016/j.jpdc.2014.03.009

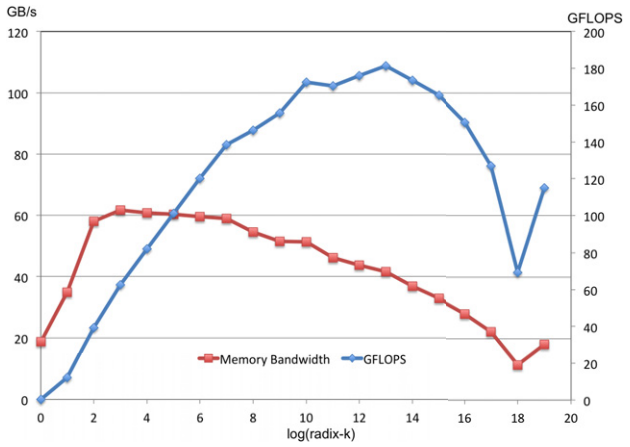


Fig. 1. Performance of batched 1D DFT using MKL library.

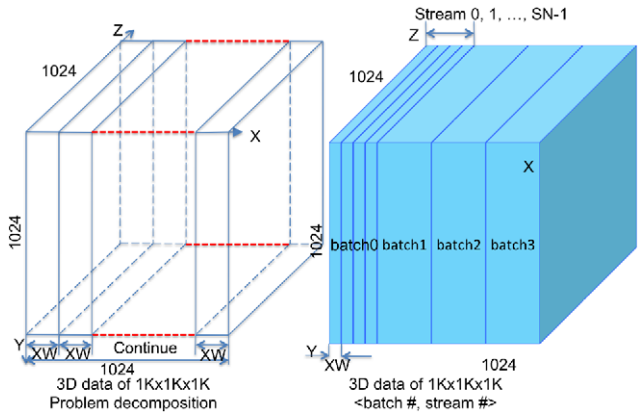


Fig. 2. 3D data memory layout.

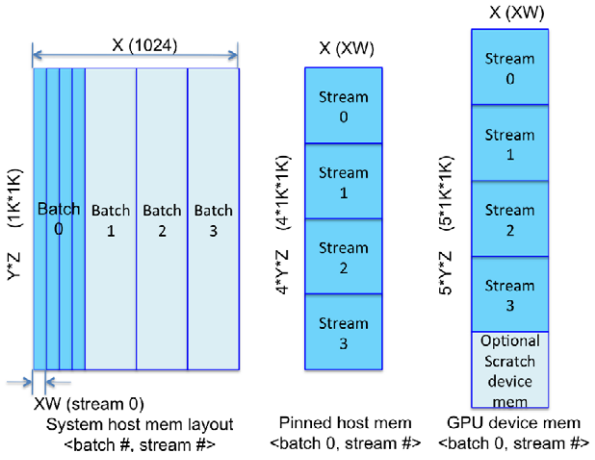


Fig. 3. CPU and GPU device memory usage.

Kernel Library (MKL) SIMD OpenMP based DFT routines to execute this step. This library seems to effectively exploit the multicore architecture, the memory hierarchy, and the SIMD capability of the core processors. As an example, we demonstrate the performance of this library on batches of one-dimensional FFTs of sizes ranging from 2^0 up to 2^{19} on the CPU of the *Sandy–Kepler* node. The results are shown in Fig. 1, where the performance is illustrated through two curves – one showing the GFLOPS performance and the second showing the memory bandwidth achieved as a function of the input size assuming only one memory read and store were done for each element. As can be seen from this figure, the memory bandwidth achieved is quite good (relative to the peak of 79.55 GB/s reported in [10]), especially in the range we are interested in (between 1 and 4 K). While the GFLOPS performance varies over a relatively significant range, it is quite good over the range of interest to us (between 1 and 4 K). Therefore, the forward and the inverse FFTs along the X dimension are completed by calling the MKL library.

3.1.2. Asynchronous streams of data movements and GPU kernels

CUDA allows the use of streams for asynchronous memory copy and concurrent kernel executions to hide long PCIe bus latency [16]. 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. The H2D and D2H memory copies can be done concurrently on the Kepler GPUs but only one-directional memory copy can be executed at a time on the Tesla GPUs. This would result in a slightly different organization of the CPU and GPU pipeline on each platform.

We now focus on the *Nehalem–Tesla* node and later address the streams used for the *Sandy–Kepler* node. In order to make effective use of the asynchronous CPU–GPU memory copy for our running example, we organize the remaining FFT computations into four batches, each consisting of four asynchronous streams where each stream involves a subarray of size $64 \times 1024 \times 1024$ (0.5 GB)—this means a vector size of 64 along the X dimension, which is demonstrated as "*XW*" in Fig. 2. The choice of vector size 64 is determined to optimize the use of the PCIe bus bandwidth. The corresponding memory layout of the problem decomposition is shown in Fig. 2.

For our running example, staging page-locked host memory of size 2 GB (0.5 GB*4) is allocated to enable asynchronous memory copy, as indicated in Fig. 3. By default, page-locked host memory is allocated as cacheable and *write-combining* flag can be used to enable the memory not to be snooped at during data transfer across the PCIe bus, which can boost the host to device bandwidth in practice [16]. However, the bandwidth on the opposite transfer direction is prohibitively slow. So we allocate two scratch page-locked memories: one with default flag and using for device to host transfer and one with *write-combining* flag and using for host to device transfer.

Fig. 4 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 $64 \times 1024 \times 1024$ along the X, Y and Z dimensions, respectively. This corresponds to the system host memory layout versus ⟨batch#, stream#⟩ in Fig. 5, which indicates 1 K × 1 K lines of 64 8-byte words with 1024 × 8-bytes stride between every two lines. Each line is denoted by *XW* in the figure, corresponding to the X-dimensional-Width.
  These apart elements need to be packed consecutively in the page-locked memory so that the following PCIe bus transfer bandwidth would be effective. The data movement for each data subset is a pipeline of block-wise movement involving a multi-threaded 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 Fig. 5.
  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

Please cite this article in press as: J. Wu, J. JaJa, Optimized FFT computations on heterogeneous platforms with application to the Poisson equation, J. Parallel Distrib. Comput. (2014), http://dx.doi.org/10.1016/j.jpdc.2014.03.009

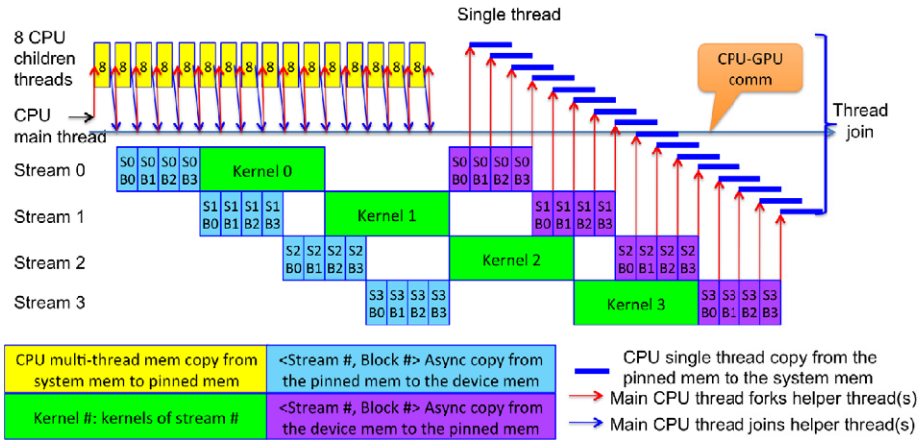


Fig. 4. CPU–GPU pipeline for *Nehalem–Tesla* node.

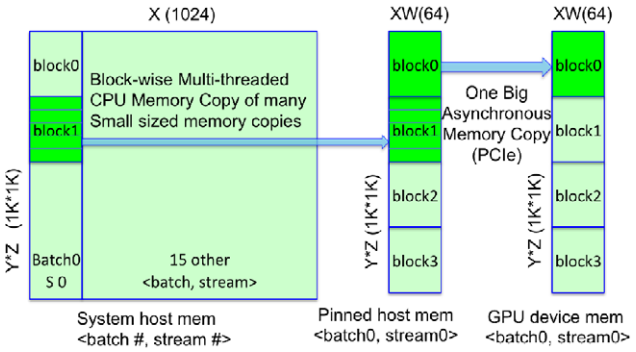


Fig. 5. Block memory copy for one stream (Tesla C1060).

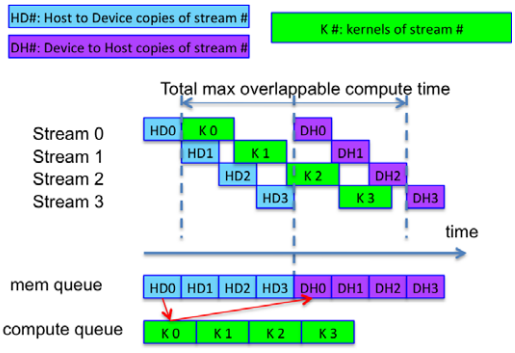


Fig. 6. Async CUDA streams for Tesla C1060.

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 execute the memory copy for one chunk of $64 \times 1024 \times 1024$ data, and launch the asynchronous kernel calls for that stream and start the same work of the next $64 \times 1024 \times 1024$ data chunk. 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 Fig. 6.

- Compute the 2D forward FFT, scaling and 2D inverse FFT computation (of 64 along the X dimension) on a chunk of size

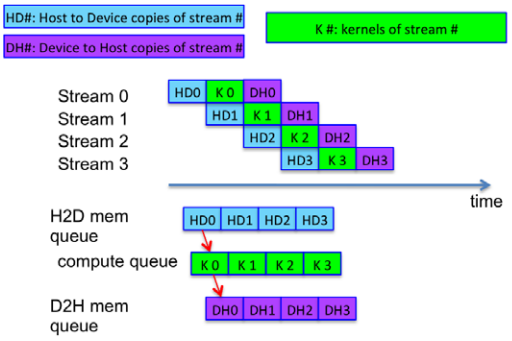


Fig. 7. Async CUDA streams for Tesla K20.

$64 \times 1024 \times 1024$ on the GPU using 7 optimized kernels. The total execution time of the kernels (of the 4 streams) should be smaller than the total transfer time of 3 streams (3 host to device and 3 device to host, Fig. 6); 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 Fig. 6. 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 [19] to compute the Y and Z dimensional FFTs of each subarray of size $64 \times 1024 \times 1024$. 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 (Fig. 3), 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.

We borrow the following notation from our earlier work [19]: $\{Y(p, q, r, n), forward\}$ amounts to the execution of p radix-q forward FFTs along the Y dimension with a stride of r with a group size of n. Using this notation, the GPU kernels can be defined by the following computations:

- $\{Y(32, 32, 32, 1024), forward\}$
- $\{Y(32, 32, 1, 32), forward\}$
- $\{Z(32, 32, 32, 1024), forward\}$
- $\{Z(32, 32, 1, 32), forward\}$
  $\{scaling, GPU\}$
  $\{Z(32, 32, 1, 32), inverse\}$
- $\{Z(32, 32, 32, 1024), inverse\}$
- $\{Y(32, 32, 1, 32), inverse\}$
- $\{Y(32, 32, 32, 1024), inverse\}$.

Please cite this article in press as: J. Wu, J. JaJa, Optimized FFT computations on heterogeneous platforms with application to the Poisson equation, J. Parallel Distrib. Comput. (2014), http://dx.doi.org/10.1016/j.jpdc.2014.03.009

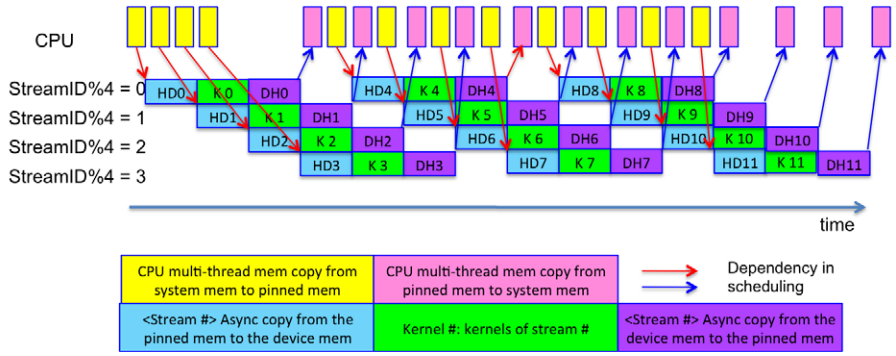


Fig. 8. CPU–GPU pipeline for *Sandy–Kepler* node.

Note that all the arithmetic computations are carried out on register contents, all global memory transfers involve coalesced memory access (the vector size along the X dimension is selected to ensure the global memory coalescing). Therefore, we complete the 64 sets of 1024×1024 forward FFT, scaling and inverse FFT using 7 kernels.

- Once the kernels are completed, we perform block-wise asynchronous memory copy from the device memory to the pinned host memory and then to the system host memory for each stream. *cudaStreamSynchronize()* is used to let the CPU memory copy back wait for the completion of the asynchronous GPU-to-CPU memory copy for that data chunk (Fig. 4).

3.1.3. Asynchronous streams of data transfers and GPU kernels for the Sandy–Kepler node

On the *Sandy–Kepler* node, memory transfers between the host memory and the device memory are possible in both directions concurrently (see Fig. 7). Therefore, rather than postpone the memory transfer of the next batch from the host memory to device memory until the completion of device memory to host memory transfer, the next batch of memory transfer could start immediately as long as the pinned host memory portion used by the same stream in the previous batch is copied into the device memory, namely, we want to ensure no overwrite hazard is possible as illustrated in Fig. 8. Only in this way bidirectional memory transfer could be maintained between batches without the pipeline being underfed—essentially we need to establish a 5-stage pipeline: (1) S2P memory copy; (2) H2D memory copy; (3) kernel execution; (4) D2H memory copy; and (5) P2S memory copy. This implies that we need at least 5 streams of data movements and GPU computations for a non-stalling pipeline.

Moreover, CUDA 5.0 added a new runtime function that allows the insertion of a callback function at any point in a stream. Such a callback function is executed on the host once all commands issued on the stream before the callback have been completed. We employ the callback for the data movement between the pinned host memory and the system host memory for that stream. The callback function for each data subset needs a private memory space to store the information about the source and destination addresses of the data subset. Because of the asynchronous execution of the memory copy and the kernel launches, we use separate space for each data subset to avoid any type of data hazards. Therefore, a straightforward implementation would be to assign each data subset to a stream, but the stream is scheduled in such a way that in the intermediate execution a fixed number of streams are active as illustrated in Fig. 8—four streams were illustrated in the figure for clarity.

In order to minimize the non-overlapping transfer time of the first and last streams, we try to reduce the size of each data subset (while still large enough to achieve high PCIe bandwidth). On the other hand, we need to guarantee the efficiency of the resultant GPU kernels—one key feature being to ensure coalesced global memory access. As 128-bytes being the largest device memory transaction size and the GPU L2 cache line size, we choose 128 bytes as the X dimension size of each data subset. As we already completed the X dimensional FFT, the choice of the 128 bytes for the X dimension is merely to optimize the GPU memory throughput performance during kernel execution. Last, we avoid the block-wise memory copy technique used in the *Nehalem–Tesla* node since each subset is already small and the overhead of blocking the subset could not be justified based on our tests.

For concreteness, let us focus on the single precision case for our running example. The entire $1\text{ K} \times 1\text{ K} \times 1\text{ K}$ dataset is divided into 64 sets, each of size $16 \times 1\text{ K} \times 1\text{ K}$ (128 MB) and organized into 64 asynchronous streams. The double precision version is merely half of the number of elements along the X dimension for each stream. The pinned host memory is large enough to hold 8 data subsets. These 64 streams are mapped into the 8 slots, eight at a time, and scheduled into execution in a round-robin order while data hazards are avoided through a shared status update protected by a MUTEX. Specifically, the possible data overwrite can only happen between streams that map to the same pinned host memory space one after another. As we are using two pinned host memory spaces for the sake of better PCIe bus bandwidth, the forward copy pinned host memory space is available for the next stream as long as the data is copied to the GPU's device memory. That is, we can proceed after the completion of the asynchronous memory copy from the pinned host memory to the GPU's device memory. As we are already using a CUDA stream callback upon the completion of the asynchronous memory copy back from the device memory to the host memory and we have enough concurrent streams ready to feed the PCIe bus, we postpone this "Green" light status update in the callback function right before the multi-threaded memory copy from the pinned host memory to the system host memory. Later, in the next round, the CPU main thread would check if the "Green" light is on before launching another streaming of copy data from the system host memory to the pinned host memory, otherwise, it would go to sleep for a while and repeat.

As a result, the GPU kernels can be defined by the following computations:

- $\{Y(4, 256, 4, 1024), \textit{forward}\}$
- $\{Y(256, 4, 1, 4), \textit{forward}\}$
- $\{Z(4, 256, 4, 1024), \textit{forward}\}$
- $\{Z(256, 4, 1, 4), \textit{forward}\}$
  $\{\textit{scaling}, \textit{GPU}\}$
  $\{Z(256, 4, 1, 4), \textit{inverse}\}$
- $\{Z(4, 256, 4, 1024), \textit{inverse}\}$
- $\{Y(256, 4, 1, 4), \textit{inverse}\}$
- $\{Y(4, 256, 4, 1024), \textit{inverse}\}$.

Please cite this article in press as: J. Wu, J. JaJa, Optimized FFT computations on heterogeneous platforms with application to the Poisson equation, J. Parallel Distrib. Comput. (2014), http://dx.doi.org/10.1016/j.jpdc.2014.03.009

3.1.4. Multi-threaded CPU inverse radix FFT computation

This step is similar to the first step—we use the MKL library to compute the X dimensional FFT with batched execution using all available cores.

4. 2 periodic 1 Neumann boundary condition case

4.1. Algorithm

Suppose our 3D input data is of size $NX \times NY \times NZ$. The 2 periodic 1 Neumann BC case involves NZ sets of 2D forward FFTs, each of size $NX \times NY$, followed by $NX \times NY$ sets of tridiagonal linear systems, each of size $NZ \times NZ$, followed by NZ sets of 2D inverse FFT of size $NX \times NY$. We use a strategy similar to the one used before to decompose the computation between the CPU and GPU while carefully organizing streams of data transfers between the two devices.

4.2. 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 for this case. The same strategy works for other problem sizes as we demonstrate later. We start with the specific details for the *Nehalem–Tesla* node, followed by the details for the *Sandy–Kepler* node.

4.2.1. Details on the Nehalem–Tesla node

- As before, the first step is carried out on the CPU, using a batch of 1D X dimensional MKL FFT library calls on all the available CPU cores.
- We launch a set of asynchronous streams involving memory copy such that each of the streams performs the following computations of data size $64 \times 1024 \times 1024$ running on the GPU:
  - Compute the forward FFTs along the Y dim: $\{Y(1, 1024, 1, 1024, \mathit{forward}, \mathit{GPU})\}$.
  - Using Thomas' algorithm, solve the tridiagonal linear systems of equations $\{Z(1024, \mathit{tridiagonal\ solver}, \mathit{GPU})\}$.
  - Compute the inverse FFT along the Y dim: $\{Y(1, 1024, 1, 1024, \mathit{inverse}, \mathit{GPU})\}$.
- After the GPU completes the execution of all the kernels and the intermediate results are written back in the CPU main memory, we execute a batch of 1D X dimensional MKL inverse FFT library calls on the available 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.

The 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: for our running example, 4 streams achieve a very good PCIe bandwidth (around 4.5 GB/s) on the *Nehalem–Tesla* node.

4.2.2. Details on the Sandy–Kepler node

On the *Sandy–Kepler* node, a similar strategy using the MKL DFT library calls is equally effective. The only difference of the 2 periodic 1 Neumann BC case from the 3 periodic BC case on the *Sandy–Kepler* node is that the Z dimensional kernels are done using different kernel functions and separate scratch space is allocated for the corresponding dataset to store vector B.

As a result, the GPU kernels can be defined by the following computations:

- $\{Y(4, 256, 4, 1024), \mathit{forward}\}$
- $\{Y(256, 4, 1, 4), \mathit{forward}\}$
- $\{Z\ \mathit{dim\ forward\ reduction}\}$
- $\{Z\ \mathit{dim\ backward\ elimination}\}$
- $\{Y(256, 4, 1, 4), \mathit{inverse}\}$
- $\{Y(4, 256, 4, 1024), \mathit{inverse}\}$.

Table 2
Compiler and library configuration.

Node	*Nehalem–Tesla*	*Sandy–Kepler*
CUDA driver	304.88	319.23
CUDA SDK	5.0	5.5
Intel compiler & MKL library	2011	2013

4.3. Arithmetic precision

When it comes to GPU performance, single precision floating point arithmetic enjoys significant benefits over double precision arithmetic [5]. 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 on many architectures, including the two GPUs we are using. An important characteristic of our algorithm is to secure a 2nd order convergence, and hence if we make the grid twice as dense, the accuracy would be four times better. 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 2nd 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 performance degradation of converting the entire data into double precision while achieving the desired accuracy.

5. Performance

In this section, we present a summary of the performance tests that have been conducted on our CPU–GPU platforms (see Tables 1 and 2).

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.

Since the essence 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 of a one dimensional FFT on data size NX is t seconds:

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

We compare the performance of our FFT implementations against implementations obtained by employing SIMD enabled OpenMP based 2D or 3D FFT routines using Intel's MKL library.

We also evaluate the effective PCIe bandwidth achieved using Formula (5) to get a sense about the performance of our CPU–GPU asynchronous streaming strategy.

$$BW = \frac{2 \cdot sizeof(element) \cdot NX \cdot NY \cdot NZ \cdot 2^{-30}}{t} \tag{5}$$

Please cite this article in press as: J. Wu, J. JaJa, Optimized FFT computations on heterogeneous platforms with application to the Poisson equation, J. Parallel Distrib. Comput. (2014), http://dx.doi.org/10.1016/j.jpdc.2014.03.009

where *sizeof* (*element*) is the number of bytes occupied by each data element—8 bytes for a single precision complex number or 16 bytes for a double precision complex number. The factor of 2 captures the fact that we are moving the data from the CPU to the GPU and then back to the CPU. The time t used in the formula excludes the CPU runtime for the X dimensional forward and inverse FFT work—it starts from the moment that 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 results are copied back into the system host memory. The performance numbers reported are the median performance of 5 runs for each data size and boundary condition combination.

5.1. The case of the 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 division scaling step in our estimate, we underestimate the performance of our algorithm. Specifically, if the total execution time on a 3D dataset of size $NX \times NY \times NZ$ is t seconds, then its GFLOPS can be measured using the standard formula:

$$GFLOPS = \frac{2 \cdot 5 \cdot NX \cdot NY \cdot NZ \cdot [\log_2 (NX \cdot NY \cdot NZ)] \cdot 10^{-9}}{t}. \quad (6)$$

The coefficient 2 in the above formula captures the forward and the inverse FFT.

Fig. 9(a) and (b) illustrate the GFLOPS performance on the *Sandy–Kepler* node of our 3 periodic BC case Poisson solver and the combined 3D forward and inverse FFT using the MKL library for the single precision and double precision cases, respectively. Fig. 10(a) shows the GFLOPS performance on the *Nehalem–Tesla* node using single precision. Due to the Tesla C1060's relatively low performance of double precision floating point operations, we did not test our algorithms on the double precision version.

For the MKL library performance on each node, the performance improved with the number of threads up to the maximum number of physical cores available on the machine. We show only the curves corresponding to the best performance on our nodes. In particular, the performance numbers of using 8 and 16 threads are similar on the dual socket quad-core *Nehalem–Tesla* node's CPU and the performance numbers of using 16 and 32 threads are similar on the dual socket octa-core *Sandy–Kepler* node's CPU—both cases achieve the peak performance of the MKL library on our platforms.

A quick comparison shows that our Poisson solver, which includes 3D forward FFT, intermediate division scaling and 3D inverse FFT almost doubles the peak performance of the MKL library on the same node. A cross comparison of the single precision version and the double precision version on the *Sandy–Kepler* node shows that the single precision version is almost double the performance of the double precision version—which indicates the robustness of our CPU–GPU workload decomposition and that our implementation is indeed limited by the PCIe bus bandwidth.

In Fig. 9(d) we illustrate the effective bidirectional PCIe bus bandwidth of the 3 periodic BC case on the *Sandy–Kepler* node: it ranges from 9 to 10 GB/s. We indicated the bidirectional bandwidth upper bound, which is the sum of pinned host memory to device memory bandwidth (5.44 GB/s) and device memory to pinned host memory bandwidth (5.34 GB/s) when the asynchronous memory copies are steady and completely overlapped. As mentioned before, when only single directional memory transfer is conducted, its performance is slightly better than concurrent memory transfer: the pinned host memory to device memory copy has a bandwidth of 5.7 GB/s and the device memory to pinned host memory has a bandwidth of 6.2 GB/s. Similarly, an average 4.5 GB/s effective PCIe bus bandwidth is achieved on the *Nehalem–Tesla* node.

Figs. 9(c) and 10(c) illustrate the effectiveness of the work decomposition on the *Sandy–Kepler* and the *Nehalem–Tesla* nodes, respectively. As we can see from these figures, the runtime of the GPU related work – including the CPU memory copy work for the GPU – constitutes more than 2/3 of the total runtime on both nodes. Recall that the execution rate of our GPU work is close to the PCIe bus bandwidth limit, and assuming more than one PCIe bus transfer is conducted, the additional runtime would surely exceed our CPU part work runtime.

5.2. The case of two periodic one Neumann boundary conditions

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

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

The number of GFLOPS needed to solve a tridiagonal linear system of size N using Thomas algorithm 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:

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

The total GFLOPS performance of our Poisson solver for this case is shown in Fig. 9(e) (single precision-SP), Fig. 9(f) (double precision-DP) and Fig. 10(b) (SP). In this case, we are comparing the performance of our algorithm to the multi-threaded CPU version implementation based on OpenMP based MKL 2D DFT routines and a fairly optimized multi-threaded tridiagonal solver. The multi-threaded CPU implementation includes the following steps: (1) NZ batched execution of the 2D forward DFT of size $NX \times NY$; (2) transpose data from memory layout of $\langle x, y, z\rangle$ to $\langle z, x, y\rangle$; (3) solve $NX \times NY$ tridiagonal linear systems, each of NZ unknowns; (4) transpose memory layout from $\langle z, x, y\rangle$ to $\langle x, y, z\rangle$; (5) NZ batched execution of the 2D inverse DFT of size $NX \times NY$. The data transpositions of steps 2 and 4 are performed to enable better memory locality for the tridiagonal solver. Otherwise, the performance will be significantly worse. In order to capture an idealized lower bound of this optimized implementation, we did not include the runtime of the memory transposition when we calculate the GFLOPS performance. Note that in reality, no matter how the boundary conditions are aligned in x, y, z dimensions, poor memory locality would be experienced in one dimension or additional memory transpositions are necessary, which would degrade the performance of the CPU implementations significantly.

As we can see from the *Sandy–Kepler* node performance figures, our single precision complete solver is significantly faster than our idealized CPU version; however such advantage decreases as we

Please cite this article in press as: J. Wu, J. JaJa, Optimized FFT computations on heterogeneous platforms with application to the Poisson equation, J. Parallel Distrib. Comput. (2014), http://dx.doi.org/10.1016/j.jpdc.2014.03.009

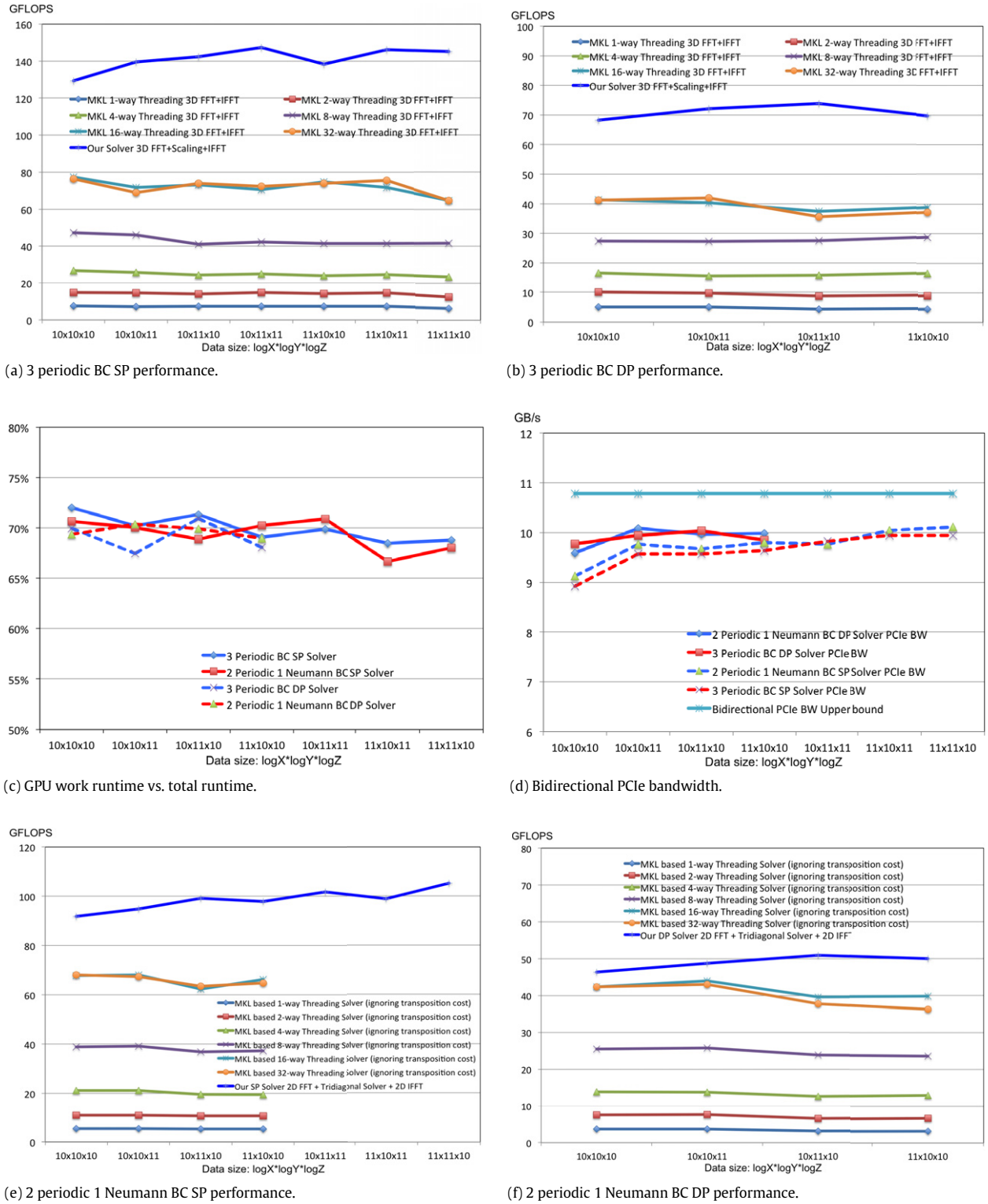
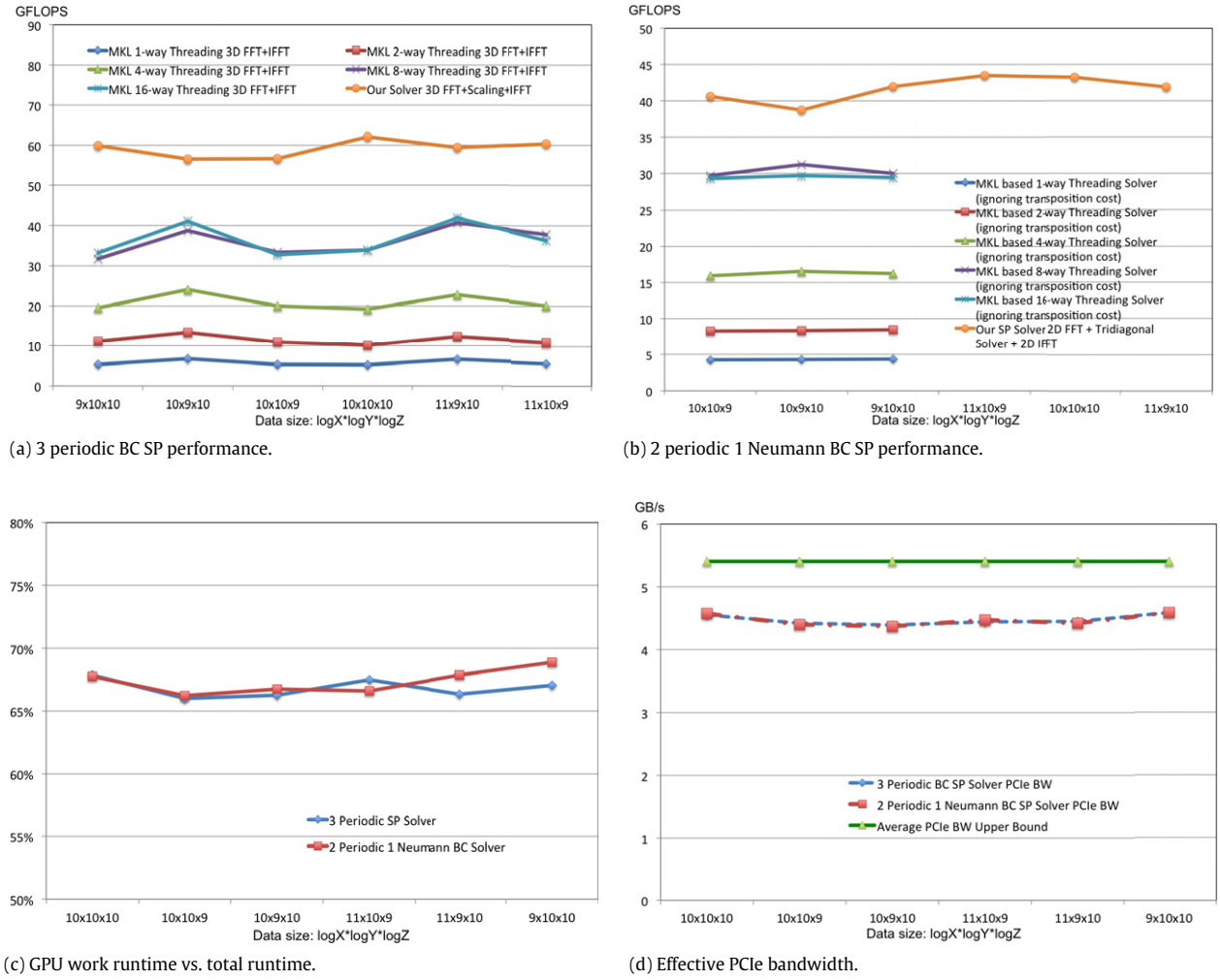


Fig. 9. Performance on the *Sandy–Kepler* node (SP: single precision, DP: double precision).

convert to double precision. The reason for the advantage degradation is because for double precision, the same amount of data was transferred by the PCIe bus with half the FLOPS computation as that of the single precision version—with the vast compute power of the GPU under-utilized. However, considering the excluded transposition time, which could be quite significant, our solvers still show superior performance as a complete solver. Our solver naturally makes use of the memory locality of the X dimension FFT computation, and carefully eliminates the need of matrix transposition when utilizing the GPU in a vector-processor way. Similar conclusion can be drawn for the *Nehalem–Tesla* node—though it is single precision version, the advantage of using GPU is degraded by the restriction of single directional PCIe bus transfer and the relatively smaller best achievable bandwidth.

Please cite this article in press as: J. Wu, J. JaJa, Optimized FFT computations on heterogeneous platforms with application to the Poisson equation, J. Parallel Distrib. Comput. (2014), http://dx.doi.org/10.1016/j.jpdc.2014.03.009



(a) 3 periodic BC SP performance.

(b) 2 periodic 1 Neumann BC SP performance.

(c) GPU work runtime vs. total runtime.

(d) Effective PCIe bandwidth.

Fig. 10. Performance on the *Nehalem–Tesla* node (SP: single precision, DP: double precision).

In terms of the PCIe bus bandwidth, Figs. 9(d) and 10(d) indicate a good PCIe bandwidth for the 2 Periodic 1 Neumann BC case—for both single and double precisions on both nodes. Moreover, Figs. 9(c) and 10(c) indicate our CPU–GPU work decomposition is quite general and effective for both the 3 periodic BC case and the 2 periodic 1 Neumann BC case.

6. 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. Our strategies were demonstrated equally effective across platforms and for different precision requirements.

Acknowledgments

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. Joseph JaJa was also supported by the National Socio-Environmental Synthesis Center (SESYNC), which is an NSF-supported center.

References

[1] Y. Chen, X. Cui, H. Mei, Large-scale FFT on GPU clusters, in: Proceedings of the 24th ACM International Conference on Supercomputing, ICS'10, ACM, New York, NY, USA, 2010, pp. 315–324. URL: http://doi.acm.org/10.1145/1810085.1810128.

[2] J. Cooley, J. Tukey, An algorithm for the machine calculation of complex Fourier series, Math. Comp. 19 (90) (1965) 297–301.

[3] Y. Dotsenko, S. Baghsorkhi, B. Lloyd, 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, ACM, New York, NY, USA, 2011, pp. 257–266. URL: http://doi.acm.org/10.1145/1941553.1941589.

[4] M. Frigo, S.G. Johnson, The design and implementation of FFTW3, Proc. IEEE (2005) 216–231.

[5] D. Goddeke, R. Strzodka, Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid, IEEE Trans. Parallel Distrib. Syst. 22 (1) (2011) 22–32. http://dx.doi.org/10.1109/TPDS.2010.61.

[6] N.K. Govindaraju, S. Larsen, J. Gray, D. Manocha, A memory model for scientific algorithms on graphics processors, in: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC'06, ACM, New York, NY, USA, 2006, URL: http://doi.acm.org/10.1145/1188455.1188549.

[7] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, J. Manferdelli, High performance discrete Fourier transforms on graphics processors, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC'08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 2:1–2:12. URL: http://dl.acm.org/citation.cfm?id=1413370.1413373.

[8] L. Gu, X. Li, 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, ACM, New York, NY, USA, 2010, pp. 305–314. URL: http://doi.acm.org/10.1145/1810085.1810127.

Please cite this article in press as: J. Wu, J. JaJa, Optimized FFT computations on heterogeneous platforms with application to the Poisson equation, J. Parallel Distrib. Comput. (2014), http://dx.doi.org/10.1016/j.jpdc.2014.03.009

[9] L. Gu, J. Siegel, X. Li, Using GPUs to compute large out-of-card FFTs, in: Proceedings of the International Conference on Supercomputing, ICS'11, ACM, New York, NY, USA, 2011, pp. 255–264. URL: http://doi.acm.org/10.1145/1995896.1995937.
[10] Intel, Intel Xeon Processor E5-2600 Product Family, 2012. https://www-ssl.intel.com/content/www/us/en/benchmarks/server/xeon-e5-hpc/xeon-e5-hpc-memory-bandwidth-stream.html?.
[11] Intel, Math Kernel Library. http://developer.intel.com/software/products/mkl/.
[12] P. Micikevicius, Multi-GPU Programming, NVIDIA CUDA Webinars. http://developer.download.nvidia.com/CUDA/training.
[13] R. Mittal, G. Iaccarino, Immersed boundary methods, Annu. Rev. Fluid Mech. 37 (2005) 239–261.
[14] A. Nukada, 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, ACM, New York, NY, USA, 2009, pp. 30:1–30:10. URL: http://doi.acm.org/10.1145/1654059.1654090.
[15] A. Nukada, Y. Ogata, T. Endo, S. Matsuoka, Bandwidth intensive 3-D FFT kernel for GPUs using CUDA, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC'08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 5:1–5:11. URL: http://dl.acm.org/citation.cfm?id=1413370.1413376.
[16] NVIDIA Corporation, NVIDIA CUDA C Programming Guide, 2013.
[17] NVIDIA Corporation, NVIDIA CUDA CUFFT Library, 2014.
[18] B.C. Sidney, Fast Fourier transforms. Appendix 1: FFT flowgraphs, 2012. http://cnx.org/content/m16352/latest/?collection=col10550/1.20.
[19] J. Wu, J. JaJa, Optimized strategies for mapping three-dimensional FFTs onto CUDA GPUs, in: Innovative Parallel Computing (INPAR), IEEE Press, 2012.
[20] J. Wu, J. JaJa, High performance FFT based Poisson solver on a CPU–GPU heterogeneous platform, in: Parallel and Distributed Processing Symposium, International, Vol. 0, 2013, pp. 115–125. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/IPDPS.2013.18.
[21] J. Wu, J. JaJa, An optimized FFT-based direct Poisson solver on CUDA GPUs, IEEE Trans. Parallel Distrib. Syst. 25 (3) (2014) 550–559.

Jing Wu received the B.Eng. degree from the Department of Electronic Information Science and Technology at the Harbin Institute of Technology, China, in 2008. She is currently a Ph.D. student in the Department of Electrical and Computer Engineering at the University of Maryland, College Park. Her area of research is high performance scientific computing and GPU acceleration.

Joseph JaJa is currently Professor of Electrical and Computer Engineering with a permanent appointment in the Institute for Advanced Computer Studies at the University of Maryland, College Park. In addition, he is the Director of Cyberinfrastructure at the National Socio-environmental Synthesis Center located in Annapolis, Maryland. Dr. JaJa received his Ph.D. degree in Applied Mathematics from Harvard University and has since published extensively in a number of areas including parallel and distributed computing, theoretical computer science, circuits and systems, and data-intensive computing. His current research interests are in high performance computing, long term management and preservation of digital information, and scientific visualization. Dr. JaJa has received numerous awards including the IEEE Fellow Award in 1996, the 1997 R&D Award for the development software for tuning parallel programs, the ACM Fellow Award in 2000, and the Internet2 IDEA Award in 2006. He served on several editorial boards, and is currently serving as a subject area editor for the Journal of Parallel and Distributed Computing and as an editor for the International Journal of Foundations of Computer Science.

Please cite this article in press as: J. Wu, J. JaJa, Optimized FFT computations on heterogeneous platforms with application to the Poisson equation, J. Parallel Distrib. Comput. (2014), http://dx.doi.org/10.1016/j.jpdc.2014.03.009