AccFFT: A library for distributed-memory 3-D FFT on CPU and GPU architectures

Amir Gholami
The University of Texas at Austin
Austin, TX
i.amirgh@gmail.com

Judith Hill
Oak Ridge National Laboratory
Oak Ridge, TN
hilljc@ornl.gov

Dhairya Malhotra
The University of Texas at Austin
Austin, TX
dhairya.malhotra@gmail.com

George Biros
The University of Texas at Austin
Austin, TX
biros@ices.utexas.edu

ABSTRACT

We present a new library for scalable 3-D Fast Fourier Transforms (FFT). Despite the large amount of work on 3-D FFTs, we show that significant speedups can be achieved for large problem sizes and core counts. The importance of FFT in science and engineering and the advances in high performance computing necessitate further improvements in existing technologies. The new library extends existing FFT libraries for x86 architectures (CPUs) and CUDA-enabled Graphics Processing Units (GPUs) to distributed memory clusters using the Message Passing Interface (MPI). Our library uses an optimized all-to-all communication for slab and pencil partitioning of both CPUs and GPUs. We present numerical results on the Maverick and Stampede platforms at the Texas Advanced Computing Center (TACC) and on the Titan system at the Oak Ridge National Laboratory (ORNL). We compare with the FFTW and the P3DFFT libraries and we show favorable performance across a range of processor counts and problem sizes. As a highlight from one of our strong-scaling experiments, our GPU-accelerated FFT is 4× faster than the P3DFFT for a 2048^3 problem on 4096 nodes on Titan using 4,096 GPUs.

1. INTRODUCTION

The 3-D Fast Fourier Transform is one of the most fundamental algorithms in computational science and engineering. It is used in turbulence simulations [19], computational chemistry and biology [8], gravitational interactions [1], cardiac electrophysiology [5], cardiac mechanics [21], acoustic, seismic and electromagnetic scattering [4, 29], materials science [22], molecular docking [18] and many other areas.

Due to its wide range of applications and the need for scalability and performance, the design of 3-D FFT algorithms remains an active area of research. Highly optimized single-node FFT algorithms have been implemented by all major hardware vendors, including Intel's MKL library [38], IBM's ESSL library [10], and NVIDIA's CUFFT [25] library. In the realm of open-source software, one of the most widely used libraries is the FFTW [13, 12]. Single-node implementations of these libraries have been extended to distributed memory versions either by the original developers or by other research groups and a large number of distributed memory libraries is currently available.

Then, what's the reason to design yet another library on FFTs? Because despite the significant amount of work that has taken place there is still room for improvement. First, many applications require strong scaling because the time-to-solution needs to be as small as possible. Typically FFT calls are invoked thousands of times within an application, for example, as part of time-stepping or Monte Carlo sampling. Often the FFT is the slowest part of the computation due to the communication costs. Therefore, small relative speedups in the FFT can result in significant absolute wall-clock gains. Second, hybrid parallelism support is not as extensive as plain MPI—especially regarding integration of MPI-based parallelism combined with accelerators. Third, the performance of the FFT can vary significantly depending on the problem size, the underlying network, and the number of processors used. Fourth, some applications require scaling to million of cores, in which case the communication costs significantly limit scalability. In a nutshell, FFT libraries need to deliver scalability from a single MPI rank to 100s of thousands of MPI ranks, while using all available system resources. This poses significant challenges to the design and implementation of efficient 3-D FFTs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS '15 June 8-11, Newport Beach, California, USA.
Copyright 2015 ACM X-XXXXX-XX-X/XX/XX ...$15.00.

Contributions.

We present AccFFT, a library that given an $N_0 \times N_1 \times N_2$ matrix of values computes its 3-D Fourier Transform. The library supports the following features:

- hybrid MPI and OpenMP parallelism,
- hybrid MPI and CUDA parallelism,
- slab (or 1D) and pencil (or 2D) decomposition,
- multiple inter-node transpose operations including an custom MPI all-to-all exchange algorithm,
- and real-to-complex (R2C) and complex-to-complex (C2C) transforms.

AccFFT uses CUFFT and FFTW for the FFT computations. AccFFT extends the single-node version of these two libraries to pencil decomposition for distributed memory FFTs. Both decompositions are necessary in order to ensure good performance across a range of MPI ranks. Slab decomposition limits the number of MPI ranks P to be less or equal to $N_0 = \max\{N_0, N_1, N_2\}$, and thus is not as scalable as the pencil decomposition. Typical values for N range from 100s to 10,000s. When it can be used, slab decomposition is typically faster than the pencil decomposition. When the FFT size is large, the all-to-all phase of the algorithm adversely affects the scaling and becomes the major bottleneck. Our custom all-to-all scheme improves significantly the performance of AccFFT in certain configurations. AccFFT implements the slab and pencil decompositions for both CPU and GPU architectures. The optimal configuration for a given problem size and processor count is determined automatically using a planning or setup phase.

We compare our code experimentally with P3DFFT [28] and FFTW on three different HPC platforms. We perform strong scaling on Maverick (with K40 GPUs) and Stampede (with Intel Xeon) at TACC and Titan at ORNL (with K20 GPUs). Overall, our implementation improves the scaling, and depending on the parameters can be significantly faster, up to 4× in one case. For the slab decomposition, P has to be significantly large—close to the maximum core count of the machine—for our code to make such a difference. The GPU version is always faster. For the pencil decomposition, our code employs multithreading on the CPU, and exclusive communication algorithms for GPUs to hide the overhead of moving data forth and back from the CPU. Our library is open source and available for download. (Link removed for anonymization).

Related work.

There is a vast literature on algorithms for FFTs and 3-D FFTs. Our discussion is by no-means exhaustive. We limit it on the work that is most closely related to ours and focus on the performance of few selected open-source libraries. Introductory material on distributed memory FFT can be found in [15]. Excellent discussions on complexity and performance analysis for 3-D FFTs can be found in [14] and [7].

• (*Libraries for CPU architectures.*) One of the most widely used packages for 3-D FFTs is the FFTW [13]. FFTW supports MPI using slab decomposition and hybrid parallelism using OpenMP. As mentioned above, the scalability of slab decompositions is limited. Furthermore, FFTW does not support GPUs. P3DFFT [28] extends the single-node FFTW (or ESSL) and supports both slab and pencil decompositions. P3DFFT does not use any special scheme for performing the all-to-all communication, and currently does not support hybrid parallelism [1]. P3DFFT is extensively used in numerical algorithms for partial differential equations. Another clone of FFTW is the PFFT library [30] which supports hybrid parallelism. However, it doesn't does not employ any special all-to-all communication algorithms as it uses FFTW transpose functions to perform the communication phase. Moreover, it does not support GPU acceleration. A very similar code to P3DFFT and PFFT is 2DECOMP [20] . Finally OpenFFT [9] employs special all-to-all schemes but does not support for multithreading or GPUs.

A multithreaded code (not open source) is described in [19] in the context of turbulence simulations. Like ours, this code is based on FFTW and employs single-node optimizations. To our knowledge, this code is one of the most scalable 3-D FFTs. The authors report results on up 786,432 cores on an IBM Blue Gene machine. However, the authors observe lack of scalability of the transpose for large core counts. On Stampede they started loosing scalability at 4,098 nodes. Also, this code does not support GPUs. In [33] the authors propose pencil decomposition optimizations that deliver 1.8× speed-up over FFTW. The main idea is the use of non-blocking MPI all-to-all operations that allow overlapping computation and communication. However the method does not address the scalability issues of FFTs. The authors compare FFTW, P3DFFT and 2DECOMP with their scheme. Other works that study 3-D FFTs on x86 platforms include [2, 26].

• (*Libraries for distributed-memory GPUs.*) The work presented in [24] is, to our knowledge, one the most efficient and more scalable distributed GPU implementations. It only supports slab decomposition so it cannot be scaled to large core counts. The scaling results presented in the paper are up to 768 GPUs. The authors employ special techniques to improve the complexity of the transpose and use an in-house CUDA FFT implementation. Their optimizations are specific to the infiniband-interconnect using the IBverbs library and thus is not portable. In the largest run, they observed 4.8TFLOPS for a 2048^3 problem on 786 M2050 Fermi GPUs (double precision, complex-to-complex), which is roughly 1.2% of the peak performance. With the understanding that comparisons between systems and configurations are not portable, we are getting about 2.2 TFLOPS on 128 K40 Kepler GPUs (double precision, real-to-complex, for a problem size of 1024^3), which is about 2.4% of the peak performance. For a C2C FFT, we get 7.1 TFLOPS using 4096 GPUs of Titan.

The DiGPUFFT open source library [7] is a modification of P3DFFT in which the intranode computations are replaced by CUFFT but no further optimization were per-

[1] The support for hybrid parallelism is on the to do list of P3DFFT and may be added in the future.

formed and as a result the is less than 0.7% of the peak. The DiGPUFFT code was not designed for production but for experimental validation of the theoretical analysis of the complexity of 3-D FFT and its implications to the design of exascale architectures.

• (*1D FFT and single-node libraries.*) Other works that analyze scalability of the FFT codes include the FFTE code [35] (which is part of the HPCC benchmark) that includes several optimizations but has only basic support for GPU using PGI compiler directives. In [39, 16] the authors propose schemes for single node 3-D FFTs. In [23], shared-memory multiple GPU algorithms are discussed. More specialized and somewhat machine-dependent codes are [17] and [11].

A very interesting set of papers proposes a different FFT algorithm that has lower global communication constants (it requires one as opposed to three all-to-all communications) and can be made even faster (up to 2×) by introducing an approximation error in the numerical calculations. The algorithm was introduced in [36] and its parallel implementation discussed in [27]. Now it is part of the MKL library. It currently supports 1D FFT transforms only. Parallel implementation of small FFTs is another challenging task. To get good performance one has to use size and machine specific optimizations, which are usually not portable. Optimizing FFT sizes of 32^3 and 64^3 has been addressed in [40], where a significant speedup was achieved on Anton system.

Limitations.

There are several limitations in our library. Currently we do not support pruned FFTs or additional features such as Chebyshev approximations that are supported by PFFT or P3DFFT. As we will discuss in the results section, the multithreading (using OpenMP) for the pencil decomposition is not very effective for large core counts. We do not have much control of this since this is hidden in the FFTW. Our implementation does not support inexact FFTs. Currently there is no support for hybrid floating point computation, but for larger FFTs it may be necessary.

Outline of the paper.

In the next section we briefly summarize the slab and pencil decompositions. In Section 2 we summarize our algorithms for CPU and GPU platforms. In Section 3 we present results from numerical experiments.

2. ALGORITHM

In this section we discuss the AccFFT library. First let us introduce some basic notation: f is input array, $\widehat{f}$ = 3-D FFT(f), P will denote the number of MPI tasks, N_0, N_1, N_2 will denote the size of f in x,y, and z direction, and $N = N_0 \times N_1 \times N_2$.

First we discuss the 3-D Fourier transform and the slab and pencil decomposition. The discrete 3-D Fourier transform corresponds to a dense matrix-vector multiplication. However, the computational complexity can be reduced by using Cooley-Tukey [6] algorithm to

$$5N_0N_1N_2(\log(N_0) + \log(N_1) + \log(N_2)). \qquad (1)$$

The forward FFT maps space to frequency domain and the inverse FFT maps the frequency to space domain. The algorithms are the same up to a scaling factor, and have the same complexity.

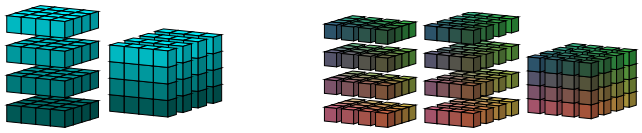

Figure 1: Decomposition of the input/output array in different stages of the forward FFT algorithm. Left:slab decomposition. Right: pencil decomposition

For many scientific applications, f does not fit into a single node and therefore the data needs to be distributed across several nodes. Two such distributions are the slab decomposition, in which the data is distributed in slabs and the pencil decomposition where each task gets a pencil of the data, as shown in Figure 1. To compute the FFT, each task has to compute its portion of FFTs, and then exchange data with other task. One can either use a binary exchange or a transpose (all-to-all) algorithm [15]. In this paper we focus on the second approach.

First we discuss the slab-decomposition, which is outlined in Algorithm 1 (forward transform) and Algorithm 2 (inverse transform), respectively. The input data is distributed in the first dimension over P task, i.e. $N_0/P \times N_1 \times N_2$, which is referred as a slab. In the forward algorithm, each task computes a N_0/P-batch of 2-D FFTs, each one having size $N_1 \times N_2$. Then an all-to-all exchange takes place to redistribute the data (indicated by the second step marked as T in Algorithm 1 and Algorithm 2. That is each task gets a

Algorithm 1: Slab decomposition forward FFT algorithm.

Input : Data in spacial domain.
Layout: $N_0/P \times N_1 \times N_2$
Output: Data in frequency domain.
Layout: $\widehat{N}_0 \times \widehat{N}_1/P \times \widehat{N}_2$

1 $N_0/P \times \widehat{N}_1 \times \widehat{N}_2 \xleftarrow{FFT} N_0/P \times N_1 \times N_2$;
2 $N_0 \times \widehat{N}_1/P \times \widehat{N}_2 \xleftarrow{T} N_0/P \times \widehat{N}_1 \times \widehat{N}_2$;
3 $\widehat{N}_0 \times \widehat{N}_1/P \times \widehat{N}_2 \xleftarrow{FFT} N_0 \times \widehat{N}_1/P \times \widehat{N}_2$;

Algorithm 2: Slab decomposition inverse FFT algorithm

Input : Data in frequency domain.
Layout: $\widehat{N}_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2$
Output: Data in spacial domain.
Layout: $N_0/P_0 \times N_1 \times N_2$

1 $N_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2 \xleftarrow{IFFT} \widehat{N}_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2$;
2 $N_0/P_0 \times \widehat{N}_1 \times \widehat{N}_2 \xleftarrow{T} N_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2$;
3 $N_0/P_0 \times N_1 \times N_2 \xleftarrow{IFFT} N_0/P_0 \times \widehat{N}_1 \times \widehat{N}_2$;

slab of size $N_0 \times \widehat{N}_1/P \times \widehat{N}_2$, where the hats denote that the Fourier Transform has been computed across the last two dimensions. To complete the transform, each task can then compute a $\widehat{N}_1/P \times \widehat{N}_2$-batch of 1-D FFTs of length N_0. The inverse FFT can be computed in a similar fashion as denoted in Algorithm 2. One advantage of our implementation is that the memory layout of the data in frequency space is the same as in the spatial one. This is different than FFTW's implementation were the default (and faster) option, changes the memory layout from xyz to yxz[2]. Here for simplicity we assumed that $N_0 = \max\{N_0, N_1, N_2\}$. In the limit of $P = N_0$, each task will just get a 2D slice of f locally. If $P > N_0$ the slab decomposition cannot be used.

An alternative is then to use the pencil decomposition method, as outlined in Algorithm 3 (forward transform) and Algorithm 4 (inverse transform), respectively. In this approach each task gets a batch of 1-D pencils, local in the last dimension. That is the memory layout of the data in each task is $N_0/P_0 \times N_1/P_1 \times N_2$, where $P = P_0 \times P_1$. We create P_0 and P_1 using MPI Cartesian communicators. We map the MPI tasks to a 2D matrix with P_0 rows and P_1 columns. To compute the forward FFT, each task first computes a $N_0/P_0 \times N_1/P_1$ batch of 1-D FFTs of length N_2. This is followed by a block row-wise all-to-all communication step in which all the tasks in the same row call exchange to collect the second dimension array locally. Then each task can compute a batched 1-D FFT of size N_1 of its local data, which is now in the form of $N_0/P_0 \times N_1 \times \widehat{N}_2/P_1$. This is followed by column-wise all-to-all exchange. A final batched 1-D FFT of length N_0 gives the forward 3-D FFT. Similar to the previous algorithm, the memory layout of the pencil decomposition is the same as in the frequency and spacial domains. However, each task owns a different pencil locally. In the Fourier space, each task owns the x pencil locally, while in spacial domain it owns the z dimension locally. This can be changed by performing two more all-to-all exchanges, but that is expensive. Instead, this exchange can happen in the inverse FFT algorithm, as given in Algorithm 4.

Algorithm 3: Pencil decomposition, forward FFT algorithm

Input : Data in spacial domain.
Layout: $N_0/P_0 \times N_1/P_1 \times N_2$
Output: Data in frequency domain.
Layout: $\widehat{N}_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2/P_1$

1 $N_0/P_0 \times N_1/P_1 \times \widehat{N}_2 \xleftarrow{FFT} N_0/P_0 \times N_1/P_1 \times N_2$;
2 $N_0/P_0 \times N_1 \times \widehat{N}_2/P_1 \xleftarrow{T} N_0/P_0 \times N_1/P_1 \times \widehat{N}_2$;
3 $N_0/P_0 \times \widehat{N}_1 \times \widehat{N}_2/P_1 \xleftarrow{FFT} N_0/P_0 \times N_1 \times \widehat{N}_2/P_1$;
4 $N_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2/P_1 \xleftarrow{T} N_0/P_0 \times \widehat{N}_1 \times \widehat{N}_2/P_1$;
5 $\widehat{N}_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2/P_1 \xleftarrow{FFT} N_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2/P_1$;

Algorithm 4: Pencil decomposition, inverse FFT algorithm

Input : Data in frequency domain.
Layout: $\widehat{N}_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2/P_1$
Output: Data in spacial domain.
Layout: $N_0/P_0 \times N_1/P_1 \times N_2$

1 $N_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2/P_1 \xleftarrow{IFFT} \widehat{N}_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2/P_1$;
2 $N_0/P_0 \times \widehat{N}_1 \times \widehat{N}_2/P_1 \xleftarrow{T} N_0 \times \widehat{N}_1/P_0 \times \widehat{N}_2/P_1$;
3 $N_0/P_0 \times N_1 \times \widehat{N}_2/P_1 \xleftarrow{IFFT} N_0/P_0 \times \widehat{N}_1 \times \widehat{N}_2/P_1$;
4 $N_0/P_0 \times N_1/P_1 \times \widehat{N}_2 \xleftarrow{T} N_0/P_0 \times N_1 \times \widehat{N}_2/P_1$;
5 $N_0/P_0 \times N_1/P_1 \times N_2 \xleftarrow{IFFT} N_0/P_0 \times N_1/P_1 \times \widehat{N}_2$;

It is well known [7] that the most expensive part of the 3-D FFT algorithm is the communication phase, which adversely affects the scaling at large core counts. This has been verified in large scale runs on Stampede and Blue waters [19]. This phase involves an all-to-all exchange, which is essentially a transpose operation between a subgroup of tasks. This exchange can be wrapped around a packing/unpacking phase to make the data contiguous in the memory. Generally the packing/unpacking phase accounts for less than 10% of the transpose time on CPU, and a negligible time on GPUs. This phase can be performed either by reshuffling of the data as done in P3DFFT, a local transpose as implemented in FFTW library, or eliminated by using MPI Data types [32]. However, we found the latter to have a poor performance, due to the overhead of sending non contiguous data. We use reshuffling of the data in the forward FFT to avoid changing the memory layout of the data in the frequency mode, and use the local transpose alternative in the inverse FFT algorithm. This is a convenient feature for users, as the data has the same ordering in memory in both the spatial domain and the frequency domains.

CPU optimizations.

The first basic optimization is that we integrate our MPI code with the multithreading option of FFTW, so we can use hybrid parallelism. Other than that the main contribution is to re-implement the pencil decomposition and optimize the all-to-all communication. The latter is the most important part of our optimization as it can deliver 2× performance gains as shown in the results section. There has been some work to optimize all-to-all, notably the Bruck algorithm [3], or a newer zero copy Bruck algorithm [37]. Here we use a modifed kway all-to-all algorithm, which provides signifant speedup for large core counts.

To optimize the all-to-all communication part, we have implemented three different communication schemes, each suitable for a range of core counts and run configurations (for example, the number of MPI tasks per node). The first option is to simply call MPI_Alltoall(v). We observed poor scaling of the transpose part for large core counts using MPI_Alltoall(v). One option is to use non-blocking point-to-point exchanges. Each task sends and receives data from all the other tasks. The all-to-all exchange finishes in p steps with a total of $\mathcal{O}(N)$ data transfer. When the number of tasks is large, network congestion can also happen, which will adversely affect the performance. As a result,

[2]FFTW3 provides an option which brings back the memory to its original format at the cost of another local transpose.

this approach is typically slower than the default MPI for large core counts. Methods such as waiting after sending a few messages can alleviate this problem, but they cannot solve the poor scaling problem. To address this problem we have adopted a custom MPI that uses the k-way recursive doubling algorithm that was presented in [34]. This all-to-all exchange sometimes alleviates the overhead associated with sending messages to a large number of tasks. In this method each task first sends the data to k other tasks, and this process gets repeated recursively. Overall in this method one transfers $\mathcal{O}(N \log p / \log k)$ data in $\log k$ steps. As one can see, the algorithm reduces to point-to-point scheme when $k = p$. The best all-to-all algorithm is selected by search during the planing phase. This is not significant since the cost can be amortized over multiple evaluations.

GPU optimizations.

For GPUs we use the CUFFT library of NVIDIA for the local FFT computations. One development was to implement a local transpose since the transpose on the NVIDIAs SDK libraries is not appropriate for the 3-D FFT since it doesn't support the correct stride. The second and the main contribution was to work around the limited bandwidth between the host CPU and GPU.

To do the all-to-all exchange we need to perform a memcpy to and from the CPU which is as expensive as the communication phase. Recently NVIDIA has introduced GPUDirect technology where GPUs on different nodes can communicate directly through the PCIe bus and avoid the CPU altogether. However, this feature requires special hardware support as well as a compatible OFED (OpenFabrics Enterprise Distribution). In the absence of GPUDirect, one option is to perform a blocking memcpy from GPU to CPU, use the transpose functions that are already implemented in the CPU code, and then copy back the results to the GPU. The packing and unpacking phases can still be performed on the GPU, as it can perform the reshuffling/local transposes much faster than CPU. Except for the MPI_Alltoall(v), it is possible to hide the extra cost of memcpy by interleaving it into send and receive operations. Instead of copying all the data at once, we divide the memcpy into chunks of the same size that each process has to send to other tasks. Each chunk is copied to a CPU buffer at a time, followed by an asynchronous send instruction. In this manner, the communication part of the CPU can start while the rest of the chunks are being copied. Since we post the asynchronous receive instructions beforehand, the receive operation can also happen with the device to host memcpy. Each received chunk can then be copied asynchronously back to the GPU.

It is not possible to perform this optimization with MPI_Alltoall as it does not provide an option to overlap the GPU to CPU and CPU to GPU memcpy. Recently, there has been some work on CUDA-aware MPI, in which one can directly pass the device pointer. For simple instructions, this can provide some speedup, but the collective operations are still not efficient. For this reason, we modified the k-way all-to-all scheme, so that it would allow for passing a device pointer. In the first phase of the algorithm, we interleave device to CPU mempcpy with send/recv operations. For the rest of the $\log k$ steps, except for the last one, only CPU memcpy will be used as the send/recv happens between CPU buffers. However, in the last step we do an interleaved memcpy from CPU to GPU similar to the point-to-point method discussed above, to maximize the overlap.

Complexity analysis.

The communication cost of AccFFT is given by $\mathcal{O}(\frac{N}{\sigma(p)})$, where $\sigma(p)$ is the bisection bandwidth of the network; for a hypercube it is $p/2$[28]. The total execution time of one 3-D FFT on a hypercube can be approximated by:

$$T_{\text{FFT}} = \mathcal{O}\left(\frac{N \log N}{P}\right) + \mathcal{O}\left(\frac{N}{P}\right).$$

The first term represents the computation and the second the memory and communication costs. For a 3-D torus topology (such as the one used on Titan) the complexity becomes

$$T_{\text{FFT}} = \mathcal{O}\left(\frac{N \log N}{P}\right) + \mathcal{O}\left(\frac{N}{P^{2/3}}\right).$$

For the GPU version this should also include the device-host communication costs. In [7] the authors give a detailed analysis in which cache effects and the local and remote memory bandwidth for GPUs and CPUs is taken into account. The basic point is that in strong scaling, the computation part becomes negligible and the overall wall-clock time will be dominated by the communication costs.

3. NUMERICAL EXPERIMENTS

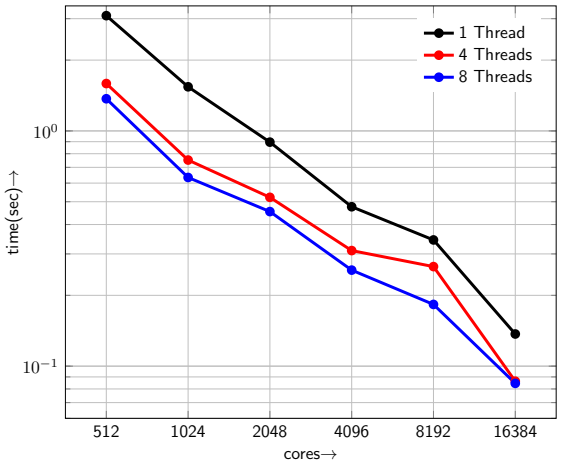

Figure 2: *CPU strong scaling results of AccFFT using the slab decomposition algorithm with a problem size of* 2048^3 *on Stampede. Wall-clock time is given for different core counts for a forward R2C FFT.*

In this section we report on the performance of AccFFT and give details regarding our implementation and the different problem sizes used for the evaluation of the library.

Computing Platforms.

We tested AccFFT on the following computing platforms:

- The **Stampede** system at TACC is a linux cluster consisting of 6400 compute nodes, each with dual, eight-core processors for a total of 102,400 available CPU-cores. Each node has two eight-core 2.7GHz Intel Xeon E5 (Sandy Bridge) processors with 2GB/core of memory and a three-level cache. Stampede has a 56GB/s FDR Mellanox InfiniBand network connected in a fat tree configuration.
- The **Maverick** system at TACC is a linux cluster with 132 compute nodes, each with dual 10-core 2.8GHz Intel Xeon E5 (Ivy Bridge) processors with 13GB/core of memory equipped with FDR Mellanox InfiniBand network.
- The **Titan** system is a Cray XK7 supercomputer at ORNL. Titan has a total of 18,688 nodes consisting of a single 16-core AMD Opteron 6200 series processor, for a total of 299,008 cores. Each node has 32GB of memory. It is also equipped with a Gemini interconnect and 600 terabytes of memory across all nodes. In addition, all of Titan's 18,688 compute nodes contain an NVIDIA Tesla K20 graphics processing unit.

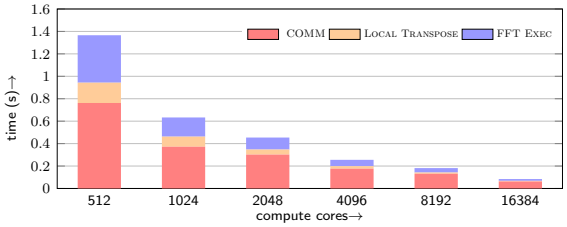

Figure 3: *Breakdown of AccFFT CPU timings in terms of communication, local transpose and local FFT times (this problem corresponds to the R2C FFT of Fig. 2 with 8 OpenMP threads).*

Implementation Details.

All algorithms described in this work were implemented using C++ using OpenMP and MPI. The only external libraries used where the MPI, FFTW, and CUFFT. On Titan we used the GCC compiler and the CRAY-MPICH libraries. On Stampede and Maverick we used the Intel compilers and the Intel MPI library. We must also mention that we compiled all the libraries with the `FFTW_MEASURE` planner flag. This flag is used to tune the library to the machine used. All results were computed in double precision. In our comparisons, we used the P3DFFT version 2.7.2 and the FFTW version 3.3.4. In both Titan and Maverick we use CUDA version 6.

Parameters in the Experiments.

The parameters in our runs are the problem size N_0, N_1, N_2, the number of tasks P, the type of all-to-all primitive used in the transpose (chosen automatically by direct evaluation), the type of decomposition (slab vs pencil),

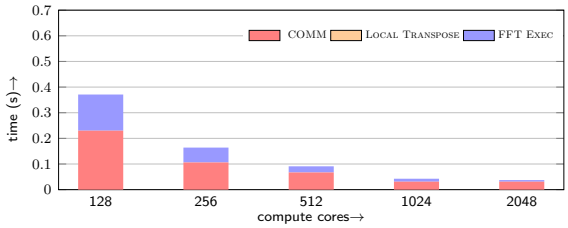

Figure 4: *Breakdown of AccFFT GPU timings in terms of communication, local transpose and local FFT. Times are given for slab decomposition of a forward R2C FFT of size* 1024^3.

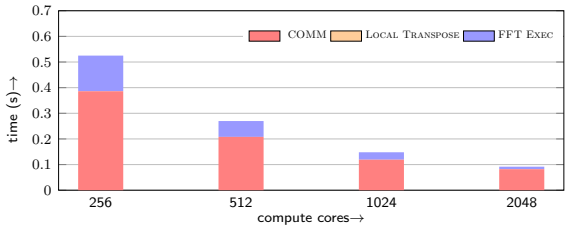

Figure 5: *Breakdown of AccFFT GPU timings in terms of communication, local transpose and local FFT. Times are given for slab decomposition of a forward C2C FFT of size* 1024^3.

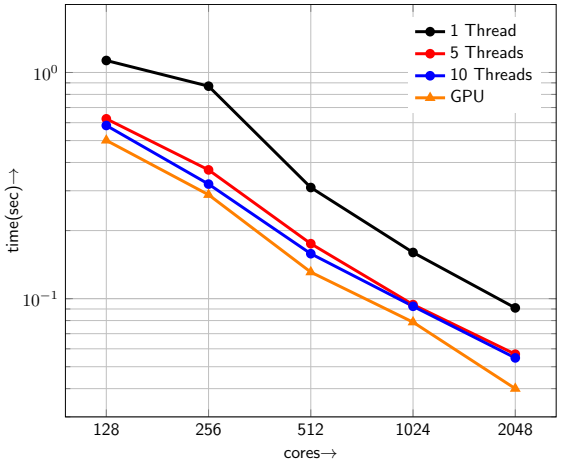

Figure 6: *Strong scaling results for AccFFT GPU and CPU. Timings are given for forward R2C FFT of size* 1024^3 *performed on Maverick.*

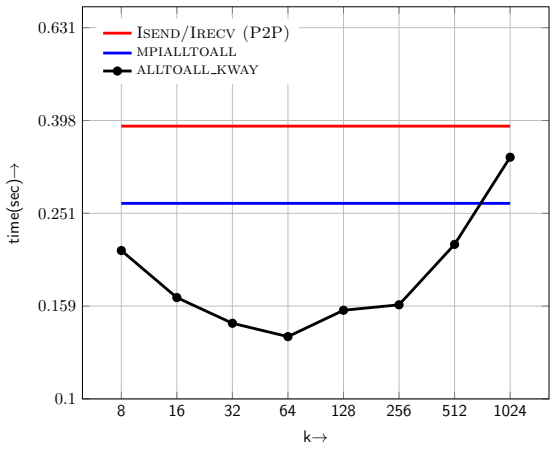

Figure 7: *Times for performing a transpose of size* 2048^3 *on 16384 cores of Stampede, Fig. 2. The timings are given for different splitting parameter, k in our kway all-to-all (denoted by* ALLTOALLV_KWAY*). The optimal transpose time corresponds to* $k = 64$ *which is two times faster than* MPI_ALLTOALL *and three times faster than performing a naive point-to-point all-to-all.*

and the use of CPU or GPU. We focus on strong scaling experiments. In most of the tests we use $N_0 = N_1 = N_2$. In all our runs, we use one MPI task per NUMA socket. In all platforms this corresponds to two MPI tasks per node. One other option can be to use more MPI procs per node. However, for the slab decomposition this further limits its maximum core counts, while multithreading allows for effective use of all the cores. If the number of threads is not specified, it is equal to the number of cores per NUMA socket. We use R2C to denote real-to-complex FFTs and C2C to denote complex to complex. Roughly speaking, the C2C transform has double the computation and communication costs of the R2C. All timings are in seconds.

Experiments.

First we examine the performance of our code on the TACC systems, then we discuss the scalability of our code on Titan, and then we conclude with a comparison with FFTW and P3DFFT on Stampede.

• In the first experiment (Figure 2) we study the strong scaling of AccFFT for a problem of 2048^3 from 512 to 16,384 cores corresponding to 32 and 1024 nodes of Stampede, respectively. We observe that OpenMP delivers 2× speed up over the sequential version. Overall we observe good scalability. In Figure 3 we break down the timings to local and global communication and computation. The communication quickly dominates the overall cost of the computation. The parallel efficiency from 512 to 16K cores is approximately 55% indicating that we could push the core count further and still get good speedup.

• The above experiment is repeated on Maverick using the GPU code for both R2C and C2C transforms of size 1024^3. We report the results of this test in Figure 6 in which we compare our CPU and GPU implementations. We break down the timings for the fastest variant in figures 4 and 5.

Table 1: *Strong scaling results for AccFFT performed on Titan. Total time is given for the slab and pencil decomposition methods for* 2048^3*. All results are for GPU C2C forward FFT. The breakdown of the total time in terms of local FFT computations, packing and unpacking (local transpose), and the communication times are given. The top four rows show the results for the slab decomposition method up to its theoretical scaling limit which is 2048 MPI tasks. The corresponding results for the pencil decomposition are given on the bottom six rows, as well as runs with higher processor counts. Note that to convert GPU count to core count, we need to multiply it by 16. So a 128 GPU run should be compared with 2048-core CPU run.*

# GPUs	Grid	Total	Comm	FFT	(Un)Pack
128	256×1	1.13	0.994	0.137	2.84e-05
256	512×1	0.914	0.85	0.0637	3.51e-05
512	1024×1	0.588	0.559	0.0289	2.89e-05
1024	2048×1	0.415	0.403	0.0127	2.87e-05
128	16×16	3.30	3.1	0.1530	0.0519
256	16×32	1.20	1.11	0.0702	0.0193
512	32×32	0.63	0.584	0.0316	0.0125
1024	32×64	0.43	0.408	0.0148	0.0092
2048	64×64	0.24	0.228	0.0066	0.0042
4096	64×128	0.20	0.187	0.0035	0.0053

We observe that although the GPU has the additional memcpy overhead, it is still slightly faster than the CPU timings. We anticipate that this bottleneck will disappear as faster and better host-device memory integration will be used in upcoming architectures. The local transpose is extremely fast. Again communication dominates the cost.

• In our next experiment (Figure 7 and Figure 8), we demonstrate the behavior of the different all-to-all exchange algorithms for a 2048^3 problem on 16,384 cores of Stampede and 512 GPUs on Titan. We observe that the default point-to-point routine is the slowest. The MPI_ALLTOALL(V) performs slightly better, but the k-way recursive all-to-all is 3× faster than the default MPI routine.

• Now we switch to Titan, where we consider the strong scaling of the GPU and CPU versions of our code. The results are given in Tables 1 and 2 for both slab and pencil decomposition for up to 4096 GPUs and 131K cores. (Notice that the 4096 GPU run corresponds to the 32K-core run, since there is 1GPU per node and 16 cores per node on Titan.) The efficiency for the largest CPU run is 27% for a 64× increase in core count. The GPU (accounting for the fact that it is C2C) is quite faster. For R2C transform on 4096 GPUs (32k cores), the GPU time is 0.1s (half of the C2C transform). For the CPU the slab decomposition cannot be used and the pencil decomposition gives 0.368 seconds with the same core counts. Generally, the slab decomposition is slightly faster than the pencil for the same core counts but it cannot be scaled. For example, for 2048 CPU cores, slab takes 1.12 seconds and the pencil takes 2 seconds. But at 16K cores the times are pretty similar, and beyond 16K cores the slab decomposition cannot be used,

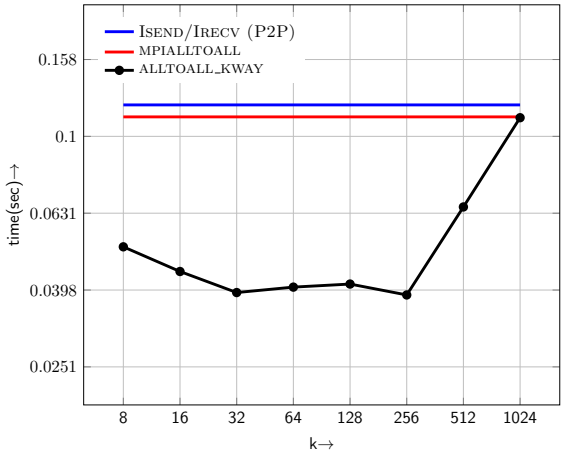

Figure 8: *Times for performing a transpose of size* 2048^3 *on cores of 512 GPUs on Maverick. The timings are given for different splitting parameter, k in the recursive all to all.*

while the pencil one scales well.

Table 2: *Strong scaling results for AccFFT performed on Titan. Total time is given for the slab and pencil decomposition methods for* 2048^3*. All results are for CPU R2C forward FFT, with 4 OpenMP threads. The breakdown of the total time in terms of local FFT computations, packing and unpacking (local transpose), and the communication time are given. The top four rows show the results for the slab decomposition method up to its theoretical scaling limit which is 2048 tasks. The corresponding results for the pencil decomposition are given on the bottom seven rows, as well as runs with higher processor counts.*

# Cores	Grid	Total	Comm	FFT	(Un)Pack
2048	256×1	1.12	0.732	0.305	0.0806
4096	512×1	0.955	0.549	0.365	0.0406
8192	1024×1	0.851	0.662	0.168	0.0206
16384	2048×1	0.401	0.298	0.0911	0.0113
2048	16×16	2.00	1.12	0.57	0.285
4096	16×32	1.14	0.846	0.151	0.142
8192	32×32	0.898	0.735	0.0798	0.077
16K	32×64	0.375	0.297	0.0377	0.037
32K	64×64	0.368	0.297	0.0376	0.014
65K	64×128	0.232	0.198	0.0202	0.009
131K	128×128	0.116	0.0974	0.0115	0.004

• In our last experiment, we compare AccFFT CPU version with P3DFFT and FFTW in Table 3. We report results for different problem sizes and processor counts on TACC's Stampede system. For the 1024^3 problem size all codes perform basically the same and multithreading offers little gains and overall the timings are very similar for all cases. For the 2048^3 problem the slab decomposition of AccFFT is more than 2× faster than P3DFFT and slightly faster than FFTW. Notice the reduced communication time. For the pencil decomposition again AccFFT is slightly faster. Running jobs larger than $16k$ cores on Stampede, requires job submission through its request queue. Therefore, we only compared AccFFT with P3DFFT for the largest run of $8,192^2 \times 4096$ on Stampede. AccFFT is roughly 2× faster than P3DFFT for the multithreaded case. However, a fair comparison would be to compared the libraries for the single thread case, where AccFFT is about 50% faster.

Table 3: *Comparison of total time, transpose, and FFT computation parts of AccFFT with the P3DFFT and FFTW libraries. We report results from tests using problem size of* 1024^3 *on 2,048 cores, problem size of* 2048^3 *on 8,192 cores, and problem size of* $8192^2 \times 4096$ *on TACC's Stampede platform. The latter was the largest run that we were able to perform by the submission deadline. All the timings are for R2C to allow comparison with P3DFFT, which does not support C2C. For FFTW and AccFFT, when we report two numbers separated by /, the first number refers to results with one thread and the second with eight threads. The Transpose operation (all-to-all exchange) does not benefit from multithreading so we report just one number. Transpose times are not reported for FFTW because they are not produced by the library. We should also mention that the FFTW runs were performed with the output in the transposed format of yxz. This saves FFTW from performing a local transpose to change the layout to xyz as implemented in AccFFT. The processor partitioning geometry for P3DFFT is reversed to account for its Fortran-based array ordering. The difference between the libraries is negligible for the problem of size* 1024^3 *on 2048 cores. However, the difference becomes more noticeable for the larger problems.*

Library	Grid	Total	Transpose	FFT
1024^3, 2048 Cores:				
AccFFT	256×1	0.101/0.084	0.045	0.059/0.037
P3DFFT	1×256	0.110	0.045	0.066
FFTW	256×1	0.125/0.111	—	—
AccFFT	16×16	0.120/0.110	0.056	0.053/0.045
P3DFFT	16×16	0.131	0.052	0.078
2048^3, 8192 Cores:				
AccFFT	1024×1	0.344/0.183	0.132	0.137/0.04
P3DFFT	1×1024	0.421	0.255	0.167
FFTW	1024×1	0.393/0.300	—	—
AccFFT	32×32	0.283/0.194	0.146	0.118/0.03
P3DFFT	32×32	0.303	0.133	0.17
$8192^2 \times 4096$, 32k Cores:				
AccFFT	4096×1	3.10/2.21	1.71	1.38/0.487
P3DFFT	1×4096	4.16	2.04	2.12

4. CONCLUSIONS

We have presented and experimentally tested AccFFT, a library for distributed memory FFTs with several unique features not found in one single library: distributed GPU calculations, multiple all-to-all variants for performance tuning, communication overlap using pipelining for the GPU version, and support for real and complex transforms, for slab and pencil decompositions.

For the slab decomposition, if P is small and there is a lot of work per MPI task, multithreading helps. The gains depend on the specific configurations. For large P, our custom k-way all-to-all is activated and when used, can reduce the communication time up to $2\times$ or $4\times$. For the pencil decomposition, both P3DFFT and CPU-AccFFT perform well. The communication scalings on Titan indicate that we don't obtain linear scaling of the communication cost whereas the computation complexity is much more consistent. Note that these results are not averaged over multiple runs; a proper analysis to estimate the constants would require multiple runs.

We remark that comparisons between systems can be hard especially if one wants to infer generalizations to other systems. On Titan the host is a bit slower so the gains from GPU are significant. On Maverick there is not so much difference, as it has faster CPUs.

Overall we presented results for the largest distributed GPU calculation (4,096 GPUs)[3], and we scaled the CPU code to 131K cores on Titan.

This work is by no means complete with these results. We are currently working on coupling the tuning process with the theoretical model developed in [7]. Using non-blocking collectives can further accelerate the calculations, if the user interleave other computations while the FFT communication is completing. The new generation of accelerators will have better memory integration and we are working to include the Xeon Phi accelerator. AccFFT can also be used in conjunction with other libraries that implement the butterfly algorithm for gerenal functions [31]. This would allow fast computation of not only the Fourier transform, but also other kernels as well.

[3]By the time of final submission, if the paper gets accepted, we will have results for 8,198 and 16,396 GPUs since our CPU code has already ran to this scale on Titan.

References

[1] S. Aarseth. *Gravitational N-body simulations*. Cambridge Univ Pr, 2003.

[2] O. Ayala and L.-P. Wang. Parallel implementation and scalability analysis of 3-D fast Fourier transform using 2d domain decomposition. *Parallel Computing*, 39(1):58–77, 2013.

[3] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(11):1143–1156, 1997.

[4] O. P. Bruno and L. A. Kunyansky. A fast, high-order algorithm for the solution of surface scattering problems: Basic implementation, tests, and applications. *Journal of Computational Physics*, 169:80–110, 2001.

[5] A. Bueno-Orovio, V. M. Pérez-García, and F. H. Fenton. Spectral methods for partial differential equations in irregular domains: the spectral smoothed boundary method. *SIAM Journal on Scientific Computing*, 28(3):886–900, 2006.

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

[7] K. Czechowski, C. Battaglino, C. McClanahan, K. Iyer, P.-K. Yeung, and R. Vuduc. On the communication complexity of 3-D FFTs and its implications for exascale. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 205–214, New York, NY, USA, 2012. ACM.

[8] R. O. Dror, J. Grossman, K. M. Mackenzie, B. Towles, E. Chow, J. K. Salmon, C. Young, J. A. Bank, B. Batson, M. M. Deneroff, J. S. Kuskin, R. H. Larson, M. A. Moraes, and D. E. Shaw. Exploiting 162-Nanosecond End-to-End Communication Latency on Anton. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, Nov. 2010.

[9] T. V. T. Duy and T. Ozaki. A decomposition method with minimum communication amount for parallelization of multi-dimensional FFTs. *Computer Physics Communications*, 185(1):153 – 164, 2014.

[10] S. Filippone. The IBM parallel engineering and scientific subroutine library. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 199–206. Springer, 1996.

[11] F. Franchetti, Y. Voronenko, and G. Almasi. Automatic generation of the HPC challenge global FFT benchmark for Blue Gene/P. In *High Performance Computing for Computational Science-VECPAR 2012*, pages 187–200. Springer, 2013.

[12] M. Frigo and S. G. Johnson. FFTW home page. http://www.fftw.org, 2000.

[13] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[14] H. Gahvari and W. Gropp. An introductory exascale feasibility study for FFTs and multigrid. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–9. IEEE, 2010.

[15] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *An Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison Wesley, second edition, 2003.

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

[17] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur, and D. K. Panda. High-performance and scalable non-blocking all-to-all with collective offload on infiniband clusters: a study with parallel 3-D FFT. *Computer Science-Research and Development*, 26(3-4):237–246, 2011.

[18] D. Kozakov, R. Brenke, S. R. Comeau, and S. Vajda. Piper: An FFT-based protein docking program with pairwise potentials. *Proteins: Structure, Function, and Bioinformatics*, 65(2):392–406, 2006.

[19] M. Lee, N. Malaya, and R. D. Moser. Petascale direct numerical simulation of turbulent channel flow on up to 786k cores. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 61. ACM, 2013.

[20] N. Li and S. Laizet. 2DECOMP & FFT-a highly scalable 2d decomposition library and FFT interface. In *Cray User Group 2010 conference*, pages 1–13, 2010.

[21] D. M. McQueen and C. S. Peskin. Shared-memory parallel vector implementation of the immersed boundary method for the computation of blood flow in the beating mammalian heart. *Journal Of Supercomputing*, 11(3):213–236, 1997.

[22] J. C. Michel, H. Moulinec, and P. Suquet. Effective properties of composite materials with periodic microstructure: a computational approach. *Comput. Methods Appl. Mech. Engrg*, 172(1–4), 1999.

[23] N. Nandapalan, J. Jaros, A. P. Rendell, and B. Treeby. Implementation of 3-D FFTs across multiple GPUs in shared memory environments. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on*, pages 167–172. IEEE, 2012.

[24] A. Nukada, K. Sato, and S. Matsuoka. Scalable multi-GPU 3-D FFT for TSUBAME 2.0 supercomputer. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–10. IEEE, 2012.

[25] C. Nvidia. CUFFT library, 2010.

[26] D. Orozco, E. Garcia, R. Pavel, O. Ayala, L.-P. Wang, and G. Gao. Demystifying performance predictions of distributed FFT 3-D implementations. In *Network and Parallel Computing*, pages 196–207. Springer, 2012.

[27] J. Park, G. Bikshandi, K. Vaidyanathan, P. T. P. Tang, P. Dubey, and D. Kim. Tera-scale 1d FFT with low-communication algorithm and intel® Xeon Phi coprocessors. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 34. ACM, 2013.

[28] D. Pekurovsky. P3DFFT: A framework for parallel computations of Fourier transforms in three dimensions. *SIAM Journal on Scientific Computing*, 34(4):C192–C209, 2012.

[29] J. Phillips and J. White. A precorrected-FFT method for electrostatic analysis of complicated 3-d structures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(10):1059–1072, 1997.

[30] M. Pippig. PFFT: An extension of FFTW to massively parallel architectures. *SIAM Journal on Scientific Computing*, 35(3):C213–C236, 2013.

[31] J. Poulson, L. Demanet, N. Maxwell, and L. Ying. A parallel butterfly algorithm. *SIAM Journal on Scientific Computing*, 36(1):C49–C65, 2014.

[32] M. Schatz, J. Poulson, and R. A. van de Geijn. Parallel matrix multiplication: A systematic journey. *(submitted to) SIAM Journal on Scientific Computing*, 2014.

[33] S. Song and J. K. Hollingsworth. Scaling parallel 3-D FFT with non-blocking MPI collectives. In *Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '14, pages 1–8, Piscataway, NJ, USA, 2014. IEEE Press.

[34] H. Sundar, D. Malhotra, and G. Biros. HykSort: A new variant of hypercube quicksort on distributed memory architectures. In *ICS'13: Proceedings of the International Conference on Supercomputing, 2013*, 2013.

[35] D. Takahashi. Implementation of parallel 1-d FFT on GPU clusters. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 174–180, Dec 2013.

[36] P. T. P. Tang, J. Park, D. Kim, and V. Petrov. A framework for low-communication 1-D FFT. *Scientific Programming*, 21(3):181–195, 2013.

[37] J. L. Träff, A. Rougier, and S. Hunold. Implementing a classic: zero-copy all-to-all communication with mpi datatypes. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 135–144. ACM, 2014.

[38] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.

[39] J. Wu and J. JaJa. Optimized strategies for mapping three-dimensional FFTs onto CUDA GPUs. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12. IEEE, 2012.

[40] C. Young, J. A. Bank, R. O. Dror, J. Grossman, J. K. Salmon, and D. E. Shaw. A 32x32x32, spatially distributed 3d fft in four microseconds on anton. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–11. IEEE, 2009.