Re-Introduction of Communication-Avoiding FMM-Accelerated FFTs with GPU Acceleration

M. Harper Langston, Muthu Baskaran, Benoît Meister, Nicolas Vasilache and Richard Lethin
Reservoir Labs Inc.
New York, NY 10012
Email: {langston,baskaran,meister,vasilache,lethin}@reservoir.com

***Abstract*—As distributed memory systems grow larger, communication demands have increased. Unfortunately, while the costs of arithmetic operations continue to decrease rapidly, communication costs have not. As a result, there has been a growing interest in communication-avoiding algorithms for some of the classic problems in numerical computing, including communication-avoiding Fast Fourier Transforms (FFTs). A previously-developed low-communication FFT, however, has remained largely out of the picture, partially due to its reliance on the Fast Multipole Method (FMM), an algorithm that typically aids in accelerating dense computations. We have begun an algorithmic investigation and re-implementation design for the FMM-accelerated FFT, which exploits the ability to tune precision of the result (due to the mathematical nature of the FMM) to reduce power-burning communication and computation, the potential benefit of which is to reduce the energy required for the fundamental transform of digital signal processing. We reintroduce this algorithm as well as discuss new innovations for separating the distinct portions of the FMM into a CPU-dedicated process, relying on inter-processor communication for approximate interactions, and a GPU-dedicated process for dense interactions with no communication.** [1]

I. Introduction

Communication, particularly global communication, is growing to be the most significant part of the cost of computation, especially as costs of arithmetic operations decrease more quickly than those for communication [1]. Significant effort has been made to generate new communication-avoiding algorithms for many matrix-based operations [2], and we have identified reduced-communication Fast Fourier Transforms (FFTs) as an area with potentially significant impact.

Many mature implementations of the FFT exist, including Netlib's `FFTPACK`, based on [3], containing a variety of `Fortran` FFT codes. In recent years the Fastest Fourier Transform in the West (`FFTW`) [4], based on the Cooley-Tukey algorithm, has been shown to provide the fastest implementation on a variety of platforms [5]. For parallel implementations, the classic *six-step framework* for computing the FFT involves three global `MPI_Alltoall` calls [6], [7], [8]. As efforts to reduce communication loads and energy consumption have attracted greater attention (e.g. [2], [9]), reducing the number of all-to-all calls in the parallel FFT has been an area of renewed interest. The SC'12 Best Paper Finalist [10] utilizes oversampled convolutions and localized FFTs coupled with a demodulation process to reduce the globalized communication load to a single step, significantly increasing the speed and decreasing the power consumption for a large-scale 1D FFT. However, [10] references an older low-communication FFT [11], which gained little traction upon its introduction, largely due to what we see as two main reservations at the time. Firstly, the high performance computing machinery necessary for such large-scale FFTs was not as mature, so the performance gain from lower communication was not as appealing. Secondly, [11] relies heavily on the Fast Mutipole Method (FMM) [12] algorithm, which may have caused hesitation due to its initial perceived complexity.

The first of these two concerns has clearly been answered by the excitement with which [10] was received while the FMM is now an established algorithm with a strong body of literature, being named one of the most important algorithms of the 20th [13] and 21st centuries [14]. We have therefore turned to reintroducing the older algorithm of [11] (hereafter referred to as the FMM-FFT), which has several distinct advantages; in particular, along with reducing the communication-load, the FMM allows one to specify the desired precision *a priori*, regardless of the complexity of the input data. Additionally, the nature of the FMM allows for the ability to separate the two main tasks, one of which requires little or no inter-processor communication and can be accelerated using GPUs as in [15] despite the increased computational demands of the FMM.

We begin by outlining the FFT and the FMM-FFT, followed by a sample of results to show that this approach is still competitive with the standard high-communication parallel FFT. We discuss how the computation phases of the FMM can be separated and finish with a discussion of our implementation of a computation-phase splitting approach and current results.

II. FFT and FMM-Accelerated FFT Overview

If a vector, $\mathbf{x} \in \mathcal{C}^n$, is of the form, $[x_0, x_1, ..., x_{n-1}]^T$, its Discrete Fourier Transform (DFT), $\mathbf{y} \in \mathcal{C}^n$ is the vector, $\mathbf{y} = [y_0, y_1, ..., y_{n-1}]^T$, formed by $y_k = \sum_{l=0}^{n-1} x_l \omega_n^{-kl}$, where $\omega_n = \exp(2\pi\ i/n)$ is an nth *principal root of unity*. In discussing the parallel implementations, the DFT is typically rewritten as $\mathbf{y} = F_n\mathbf{x}$, where $[F_n]_{(j,k)} = \omega_n^{-jk}$.

Computing the DFT of a vector of size n requires $O(n^2)$ total operations. The *Fast Fourier Transform* (FFT) lowers this

[1]Sponsored by Defense Advanced Research Projects Agency, Microsystems Technology Office (MTO). Program: Power Efficiency Revolution for Embedded Computing Technologies (PERFECT). Issued by DARPA/CMO under Contract No: HR0011-12-C-0123. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

time to $O(n \log n)$ [8], and while recursion can be employed in computing the FFT, most implementations instead use nested loops. Further, the fastest implementations place the input into bit-reversed order, the most popular being the *Cooley-Tukey* algorithm [8] and its many variations.

For p distributed-memory processors, $0 < p < n$, let $m = n/p$, where m is an integer such that $F_{n|p}$ signifies the top-left $m \times m$ submatrix of $F_n/\sqrt(n)$, notation specific to [11]. Based on [8] and the split-radix variation [7], the radix-p splitting presents a factorization of F_n [6]:

$$F_n = (F_p \otimes I_m)T(I_p \otimes F_m)\Pi, \tag{1}$$

where T is the diagonal matrix of twiddle factors and Π performs necessary permutations [10]. Moving right to left in Equation (1), we can compute F_n in six steps:

- **Step 1**: Global bit reversal Π;
- **Steps 2-3**: Local FFTs, global transpose $(I_p \otimes F_m)$;
- **Step 4**: Twiddle factors T;
- **Steps 5-6**: Local FFTs, global bit reversal $(F_p \otimes I_m)$.

The three globalized operations in the steps above in computing F_n require three *all-to-all* steps when data order preservation is required [10]; this common approach for column-major layouts is referred to as the *six-step framework* [6] and can be seen as a consequence of [7]'s evolution from [8]. For row-major layouts, [6] further discuss a *four-step framework* approach.

For the FMM-FFT, [11] refactors Equation (1) as

$$F_n = (I_p \otimes F_m)(F_p \otimes I_m)M\Pi, \tag{2}$$

where $M = diag(I_m, C^1, ..., C^{p-1})$ for

$$C^s_{(j,k)} = \rho^s \left[cot\left(\frac{\pi}{m}(k - j + s/p)\right) + i\right]$$

and $\rho^s = exp(-i\pi s/p)sin(\pi s/p)/m$. The C^s operators can be applied quickly with a 1D FMM [16], a process employed by [17] for serial 1D FFTs. In order to turn this into a parallel algorithm with a single all-to-all call, [11] assume $\mathbf{x}$ and $\mathbf{y}$ are stored in block order (i.e., $\mathbf{x} = [\mathbf{x}_0, ..., \mathbf{x}_{p-1}]^T$, $\mathbf{y} = [\mathbf{y}_0, ..., \mathbf{y}_{p-1}]^T$ for p processes) and perform the following:

1) In processor μ, perform $C^\mu \mathbf{x}_\mu$ in parallel (this incorporates the distributed $\Pi\mathbf{x}$ calculation);
2) Perform $m = n/p$ p-sized distributed FFT operations, corresponding to $(F_p \otimes I_m)$ (this distributed permutation cannot be avoided unlike the one in Step 1);
3) For each processor, μ, perform a local m-sized FFT, corresponding to $(I_p \otimes F_m)$.

This approach reduces the communication by nearly half, totaling $O(2p + 5\log_2 p - 8)$ messages sent per processor, assuming 15 digits of accuracy and $N^{FMM}_{leaf} = 32$ as per [11], where N^{FMM}_{leaf} is the number of points stored per leaf level in the FMM algorithm. We now discuss an overview of the FMM in Section III, followed by results.

III. FMM Overview

The FMM is a mature algorithm, so we provide only a brief overview of the structure of a basic 1D FMM with a purely uniform distribution of points. Please refer to [12], [18], [19] for more details.

Given force distribution g at N_{src} source locations, $\mathbf{y}_i$, we wish to compute the potential u at N_{trg} target locations, $\mathbf{x}_j$:

$$u(\mathbf{x}_j) = \int_{\mathbb{R}} K(\mathbf{x}_j, \mathbf{y})g(\mathbf{y})d\mathbf{y} \approx \sum_{i=1}^{N_{src}} K(\mathbf{x}_j, \mathbf{y}_i)g(\mathbf{y}_i)w_i, \tag{3}$$

where K is a kernel operator, w_i is a quadrature weight associated with $\mathbf{y}_i$, and $j = 1, \ldots, N_{trg}$. For $N_{src} = N_{trg} = N$, the FMM decreases the computational cost from $O(N^2)$ to $O(N)$ for a fixed user-prescribed level of accuracy by introducing a hierarchical tree partition of a bounding domain D, enclosing all points, and two series expansions for each subinterval at each level of the hierarchy. For the root of the tree at level $\ell = 0$, associated with D, the intervals at level $l + 1$ are obtained recursively, subdividing each interval at level l into two sub-intervals, referred to as its children.

In general, the FMM consists of two steps, denoted as the *near-field* (NF) and *far-field* (FF) computation phases. Specifically, for interval B of width H, its *near field* (NF^B) is the set of all subintervals in D contained inside an interval centered at B of width $3H$, and its *far field* (FF^B) is the complement of NF^B: $FF^B = D \setminus NF^B$. Finally, the *interaction list* of B, denoted by L^B_I, is defined to be the children of B's parent's neighbors that are not neighbors themselves such that $L^B_I \subseteq FF^B$. The depth of the tree is chosen such that the smallest intervals (leaf nodes in the tree structure) contain no more than some fixed number of points, N^{FMM}_{leaf}. For uniformly-refined trees, the total number of nodes is bounded by $2N/3N^{FMM}_{leaf}$. Thus, if the workload per interval is constant, the net algorithm has $O(N)$ complexity.

Two types of series (represented as vectors of coefficients) are associated with each interval B in the hierarchy. The length of each expansion is chosen *a priori* such that the truncated mathematical series numerically approximates the original source distribution to within ϵ^{FMM}_{tol}:

- The *multipole expansion* encodes information in B to approximate the potential at locations in FF^B;
- The *local expansion* encodes information from $V \in FF^B$, approximating the induced potential on B.

The FMM computes the total field at B as the sum of the contribution from sources in NF^B and the contribution from sources in FF^B. Contributions from NF^B are computed directly using dense summations while contributions from FF^B are obtained by evaluating the approximating expansions.

All of the tools exist for an $O(N \log N)$ method, but one can do better using the following *translation* operators:

- **Source to Multipole (S2M)** translates the source forces at a leaf interval into its multipole coefficients;
- **Multipole to Multipole (M2M)** translates the multipole coefficients of an interval's children into its own;

- **Multipole to Local (M2L)** translates the multipole coefficients of an interval into the local coefficients of a non-adjacent interval;
- **Local to Local (L2L)** translates the local coefficients of an interval's parent into its own;
- **Local to Target (L2T)** translates the local coefficients of a leaf interval into induced potentials at its targets.

For the $O(N)$ FMM, the essential task is the *construction* of the local expansion coefficients in a hierarchical manner, using an *upward pass* from the finest level to the coarsest, followed by a *downward pass*. After computing the approximate FF contributions, the final step is to compute direct NF interactions for leaf intervals. The full flow of the algorithm is outlined below in Algorithm 1.

Algorithm 1 Non-Adaptive Fast Multipole Method

STEP 1 - Construct tree T such that leaf B contains fewer than N_{leaf}^{FMM} points and set expansion lengths based on ϵ_{tol}^{FMM}.
for each B in *preoder* traversal of T **do**
  build near field NF^B and interaction list, L_I^B
end for
STEP 2 - UPWARD PASS
for each B in *postoder* traversal of T **do**
  if B is a leaf interval: Construct its multipole expansion from its source points and forces using the S2M operator.
  if B a non-leaf interval: Construct its multipole expansion from each of its children using the M2M operator.
end for
STEP 3 - DOWNWARD PASS
for each interval B in preoder traversal of T **do**
  Compute the contribution to B's local expansion from its parent using the L2L operator and from L_I^B list using the M2L operator.
end for
STEP 4 - DIRECT CALCULATIONS
for each leaf interval B in T **do**
  Compute the potential at each target location from B's local expansion using the L2T operator and from NF^B using direct calculations.
end for

IV. FMM-FFT RESULTS

The full FMM-FFT algorithm has been reimplemented in `C` with generous sharing of the original `Fortran 77` code by Dr. Alan Edelman along with improvements and optimizations. We have performed the tests in this section of the rebuilt FMM-FFT algorithm on a distributed-memory Linux OS cluster with Intel(R) Xeon(R) CPUs X5550 @ 2.67GHz CPU (8-cores per node with 24 GB of memory per node) and QDR InfiniBand networking. We have replaced the `FFTPACK` [3] package with `FFTW` [4] for the local FFTs. `MPI` is utilized for inter-node communication purposes, and the code is compiled using the `MVAPICH mpicc` wrapper for Intel's `icc` (version 11.1) with the `-O2` compilation flag.

A. Test 1: Fixed Problem Size with Varying Processors

For our first test, we compare the operation count (number of floating-point operations), the communication count (number of messages sent), and the overall runtime (total wall time in seconds) of the parallel FMM-FFT to a standard six-step parallel FFT. In Figure 1, we run a test with 1.68×10^7 points, a fixed FMM precision of 10^{-12}, and $N_{leaf}^{FMM} = 32$.

As can be seen in Figure 1 (*left*), the overall operation count for the FMM-FFT is considerably greater than the standard FFT due to many matrix-vector product function calls in the FMM. Both algorithms perform two FFTs while the FMM adds nearly a factor of 10 more operations in this current implementation. However, the communication count on each processor for the FMM-FFT consists of some small amount of communication in the FMM along with a single all-to-all call. By comparison, the standard FFT performs three all-to-all calls; hence, as seen in Figure 1 (*middle*), the communication costs for the standard FFT are much greater.

Despite the fact that the operation count for the FMM-FFT algorithm is significantly greater than for the standard FFT, the communication count is nearly three times less. The resulting timings in Figure 1 (*right*) show that the FMM-FFT wall times are either on par with or approximately twice that of the standard FFT.

By more closely investigating the operation count for the FMM algorithm alone, we see in table I that for the tests in Figure 1, approximately the same amount of operations are performed in the NF, counted in NF^{ops}, as in the FF, counted in FF^{ops}.

NP	NF^{ops}	FF^{ops}
2	2.53×10^9	2.39×10^9
4	1.86×10^9	1.47×10^9
16	5.75×10^8	4.19×10^8
32	2.97×10^8	2.14×10^8
64	1.51×10^8	1.09×10^8
128	7.60×10^7	5.66×10^7

TABLE I. FOR THE TESTS IN FIGURE 1, NEAR-FIELD VERSUS FAR-FIELD OPERATION COUNTS (NF^{ops} AND FF^{ops}, RESPECTIVELY).

The NF does not require any (or nearly no) communication due to computations in that portion relying solely on adjacent intervals. Hence, communication can be reduced in the FMM at the cost of increased NF^{ops} by increasing N_{leaf}^{FMM}; conversely, NF^{ops} can be reduced by decreasing N_{leaf}^{FMM}, thereby increasing FF^{ops} *and* overall time spent in communication (reducing the overall strength of the FMM-FFT approach). As can be seen in Step 4 of Algorithm 1, there is no overlap in the NF and FF computation phases until the L2T operator translates the FF contributions. Hence, Steps 2 and 3 (using the S2M, M2M, M2L and L2L operators) can occur at the same time as the direct NF computation in Step 4 as long the direct NF computations and L2T operations do not overlap. We discuss how we can take advantage of this property of the FMM in Section V in order to separate the processes between the CPU and GPU.

In the next test we investigate how changing N_{leaf}^{FMM} affects the computation loads between NF^{ops} and FF^{ops} as well as the effect on the running times.

B. Test 2: Fixed Problem Size with Varying Near-Field Loads

As proposed, changing N_{leaf}^{FMM} shifts the balance between NF^{ops} and FF^{ops}. Along with the numerical precision, N_{leaf}^{FMM} is the most tunable FMM parameter, and by increasing N_{leaf}^{FMM}, the height of the resulting tree structure decreases. In fact, for input of size N, a tree with height zero simply results in a fully-dense $O(N^2)$ computation while a tree with a single point per leaf interval results in a tree with unnecessary height

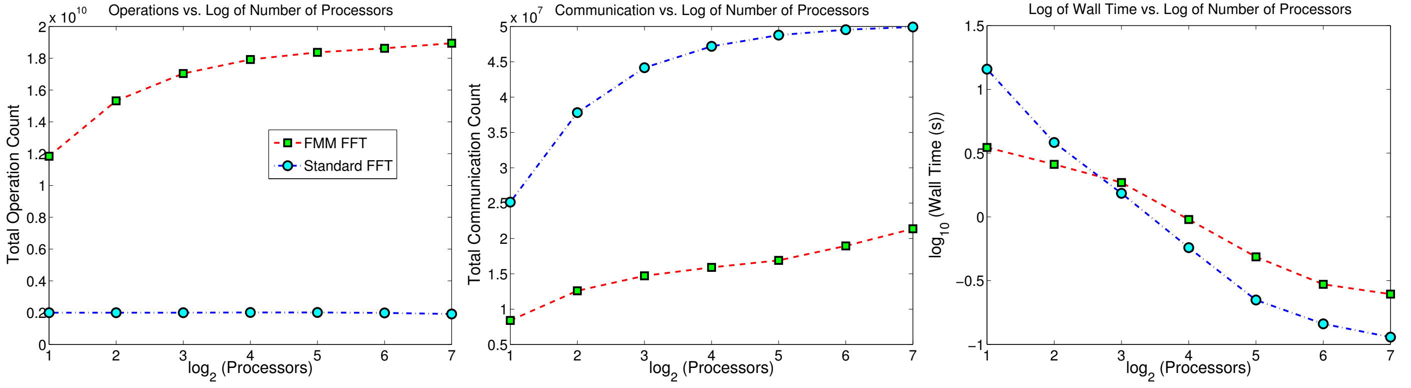

Fig. 1. FMM-FFT versus standard six-step parallel FFT for a fixed problem size on varying processors with various measurements: *Left*: Total operation count (number of floating point operations); *Middle*: Total communication count (number of messages sent); *Right*: log_{10} of wall times (in seconds).

and significantly greater FF^{ops}. In figures 2 and 3, we fix the problem size at 1.34×10^8 points, the FMM precision at 10^{-12} and the number of processors at 64, investigating NF^{ops} and FF^{ops} as N_{leaf}^{FMM} increases.

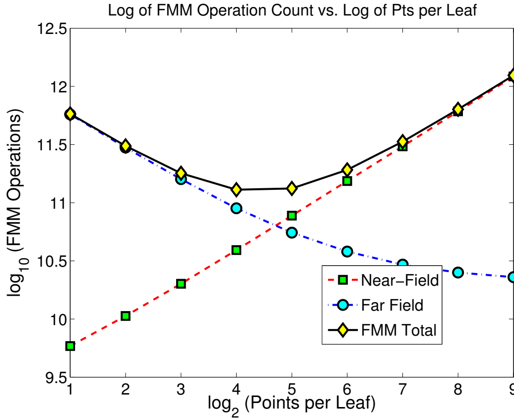

Fig. 2. Measuring effect on NF^{ops} and FF^{ops} with respect to N_{leaf}^{FMM} with 64 processors and 1.34×10^8 points.

As expected, as N_{leaf}^{FMM} increases, NF^{ops} increases significantly while FF^{ops} decreases significantly in Figure 2. Further, this has the expected effect on the wall-time seen in Figure 3 since the NF computations are a dense operation and hence more computationally intensive than FF computations.

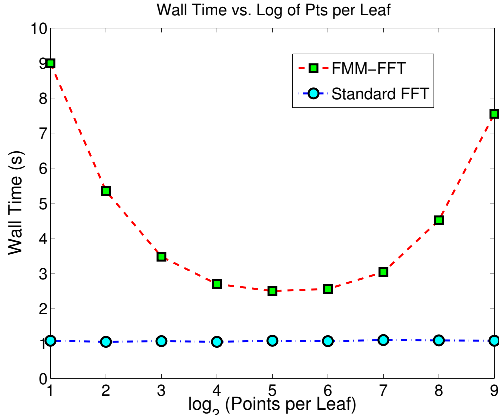

Fig. 3. Measuring effect on the wall-time for FMM-FFT with respect to N_{leaf}^{FMM} with 64 processors and 1.34×10^8 points as the computation load is shifted between the FF and NF.

These tests indicate FF^{ops} can be reduced significantly, though by increasing NF^{ops}. Additionally, we note that the wall-time for the FMM reaches an *optimal* level; this in fact concurs with [11], where it is noted that the overall computation costs are minimized when $N_{leaf}^{FMM} = -log_{10}(\epsilon_{tol}^{FMM}) \cdot \sqrt{10/3} \approx -2 \cdot log_{10}(\epsilon_{tol}^{FMM})$. In further tests not shown here, we utilized this optimal choice for N_{leaf}^{FMM}, resulting in the same conclusions in terms of measuring NF^{ops} versus FF^{ops}. In the next test, we investigate the effect of varying the FMM precision; as expected, there is a small effect on communication and FF^{ops}, but not on NF^{ops}.

C. Test 3: Fixed Problem Size with Varying FMM Accuracy

The numerical accuracy of the FMM-FFT algorithm has been separately verified for a requested level of precision in the FMM (the only portion that does introduce a level of numerical accuracy outside of machine-level precision). That is, for requested FMM precision,ϵ_{tol}^{FMM}, if $\tilde{\mathbf{y}} = \tilde{F}_n^{\epsilon_{tol}^{FMM}} \mathbf{x}$, then $||\mathbf{y} - \tilde{\mathbf{y}}|| \approx \epsilon_{tol}^{FMM}$. We do not report these results here as the focus is more on computation and communication costs and less on numerical accuracy. Further, the proofs of the numerical accuracy and stability are available in [11]. However, we are interested in the effect on the communication and operation counts when varying ϵ_{tol}^{FMM} as this affects the sizes of the expansion coefficients and translation operators in the FMM's FF computation phase (specifically Steps 2 and 3 in Algorithm 1 in Section III). In Figure 4, we again set the problem size to 1.34×10^8 points and the number of processors to 64, investigating NF^{ops} and FF^{ops} as the requested level of precision increases (or as ϵ_{tol}^{FMM} decreases).

Since ϵ_{tol}^{FMM} has a direct impact on the FF computations, Figure 4 (*left*) shows a measurable effect on the operation balance. The additional communication count in Figure 4 (*middle*) is largely negligible, but the increased operation cost leads to an expected increase in wall times in Figure 4 (*right*). For applications which require low levels of overall precision, this suggests that the FF and NF operation counts can be re-balanced to take advantage of desired ϵ_{tol}^{FMM}.

We now turn to discussing how to exploit the natural split between the NF and FF computation steps in the FMM.

V. GPU-Accelerated FMM-FFT Overview and Results

The FMM is well-tailored for parallel implementation, and [11] uses [20]'s approach with the specific kernel operator,

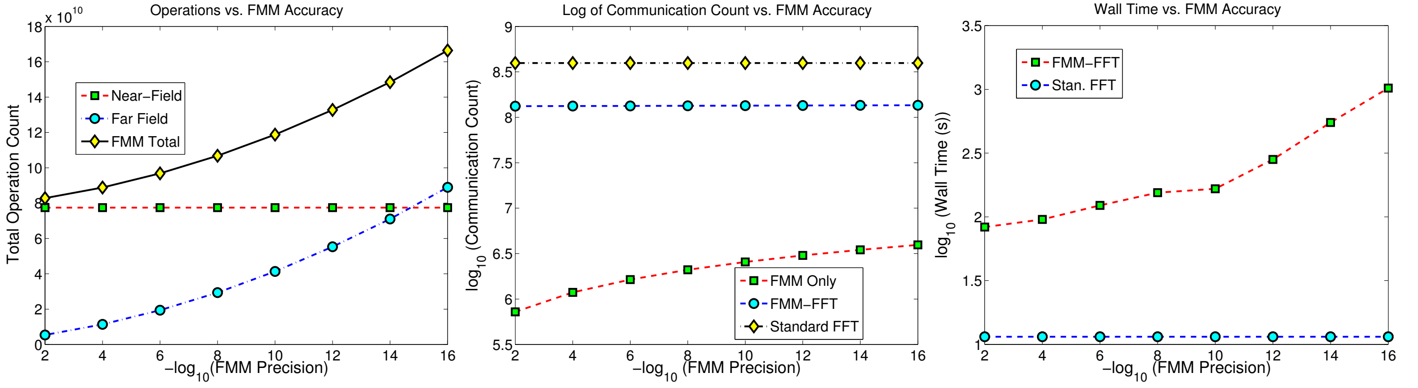

Fig. 4. Measuring the effect of FMM accuracy on the FMM-FFT algorithm: *Left*: Splitting the NF and FF operation counts shows that as expected, only the FF operation count is affected as accuracy increases; *Middle*: Total communication count for the FMM-FFT is largely unaffected by increased accuracy; *Right*: Wall times begin to show a moderate effect beyond 10 digits of requested accuracy.

$K(r) = cot(r/2)$ for Equation (3), which we additionally employ. The parallel FMM algorithm, however, has seen many improvements, most significantly [15], in which the NF and FF computation steps are split onto the GPU and CPU, respectively for a distributed memory implementation.

Using NVIDIA's `CUDA` framework to accelerate the NF computations on the GPU as in [15] has the potential for greatly shifting the load balance between the NF and FF as we can increase the number of threads in the NF computations. Additionally, for a well-designed reordering, we stand to reduce the overall power consumption [21] of the FFT, both in utilizing the GPU structure as well as the reduced communication loads of the FMM-FFT.

In Figure 5, we show a system where the FF computations are handled on the CPU while the NF computations are handled on a GPU, assuming a dedicated node for each `MPI` process and a dedicated GPU for each portion of the NF affiliated with that process.

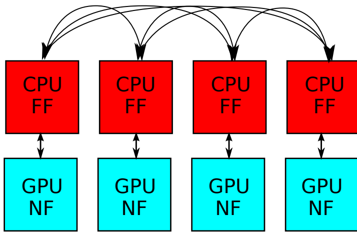

Fig. 5. Splitting the FF and NF computations onto the CPU and GPU, respectively, for the FMM portion of FMM-FFT algorithm.

The nature of the redesigned FMM-portion of the algorithm in Section II is hence as follows:

1) For a problem of size n with p processors, split the input, $\mathbf{x}$ into n/p portions and distribute;
2) For a requested number of points per leaf interval, N_{leaf}^{FMM}, and a specific processor, μ, split μ's portion (denoted as $\mathbf{x}_\mu$ into N_{leaf}^{FMM}-size subintervals;
3) Perform a small amount of communication to ensure all leaf intervals μ have access to the source data from adjacent intervals which may lie on different processors (which may occur on the endpoints of $\mathbf{x}_\mu$);
4) Asynchronously copy $\mathbf{x}_\mu$ and subinterval data from the CPU to the dedicated GPU for μ.
5) Compute the NF interactions on the GPU;
6) Compute the FF interactions and combine with the GPU contribution;
7) For $m = n/p$, perform m p-sized distributed FFT operations, corresponding to $(F_p \otimes I_m)$;
8) For each processor, μ, perform a local m-sized FFT, corresponding to $(I_p \otimes F_m)$.

Again, the NF and FF computations (Steps 5 and 6, respectively) can occur concurrently. Further, as the cost per operation on the GPU will be different than on the CPU, the data balancing between these steps can be altered to increase the dense contributions as necessary. We show initial results for the GPU-accelerated FMM-FFT as well as discuss ongoing innovations in the next section.

A. GPU-Accelerated FMM-FFT Results

We have tested the above approach on a cluster of four Xeon(R) CPU X5650 @ 2.67GHz CPUs, each with a dedicated 448 thread Tesla-M2070 GPU node. Our current GPU implementation is designed to take advantage of the size of N_{leaf}^{FMM}. That is if $N_{leaf}^{FMM} \geq 512$, the approach is maximizing the number of threads available. We begin by fixing the input size as $n = 1.34 \times 10^8$ points and the number of processors at 2 and varying the size of N_{leaf}^{FMM} from small to large (CPU/`MPI` heavy loads to GPU/`CUDA` heavy loads, respectively). The results are in Figure 6.

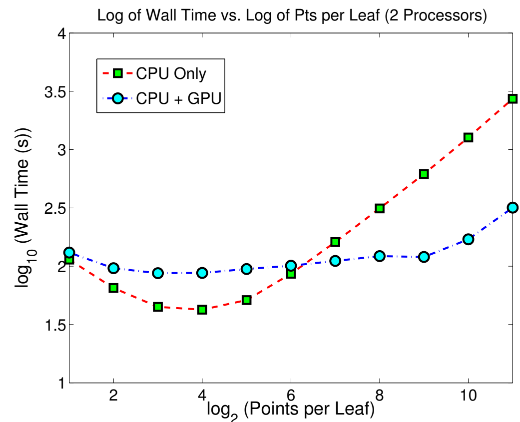

Fig. 6. Comparing CPU and CPU-GPU FMM-FFT implementations for 2 processors with respect to varying NF loads by increasing N_{leaf}^{FMM}.

As can be seen, for smaller N_{leaf}^{FMM}, the original pure `MPI` code is faster than the `MPI-CUDA` code. However, as we

increase N_{leaf}^{FMM} and the NF load, the GPU-accelerated code significantly outperforms, nearly by a factor of 10 for large N_{leaf}^{FMM}. In fact, for $N_{leaf}^{FMM} < 512$, the GPU is not being fully-utilized due to the nature of our GPU optimizations, so the improvement is more evident for larger N_{leaf}^{FMM}.

We repeat the above test with 4 processors in Figure 7.

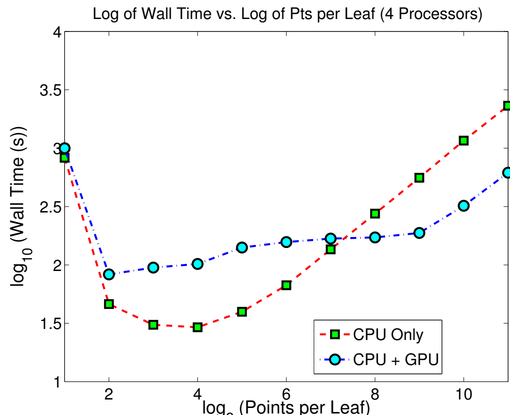

Fig. 7. Comparing CPU and CPU-GPU FMM-FFT implementations for 4 processors with respect to varying NF loads by increasing N_{leaf}^{FMM}.

Again, we see that for smaller N_{leaf}^{FMM}, the pure `MPI` code is faster, but as we achieve full GPU load for $N_{leaf}^{FMM} \geq 512$, our `MPI-CUDA` code performs significantly better.

VI. Conclusion

We have shown results for our reimplementation in `C` of the FMM-accelerated low-communication 1D FFT from [11] as well as shown how the loads are balanced between the NF and FF operations for the FMM. Reviewing the FMM and how the NF and FF contributions are separate processes, we have shown how their contributions can be asynchronously computed on the GPU and CPU, respectively. We have discussed our current implementation of this splitting process and have shown current, promising results for a small number of processors with dedicated GPUs

Ongoing work is focused on optimizing these GPU accelerations. The current implementation, due to the nature of the complex data structures, makes preserving locality difficult, resulting in potentially poor memory performance. This is especially true as the number of processors grows. As such, we are working to incorporate R-Stream [22], a High Level Compiler for embedded computing and parallel processing of algorithms, to generate higher quality implementations of the GPU portion. Further optimizations involve exploiting inherent symmetries in the FMM structure as in [19] as well as providing a pure single-precision option for the entire algorithm for additional efficiency when lower numerical accuracy is desired.

Additionally, we wish to compare this new implementation (and the six-step parallel FFT) to a four-step parallel FFT, which employs a single global transpose as described in [6], as well as additional FFT implementations. Current discussions with other research groups have focused on such collaborations. Additional discussions have involved investigating the feasibility of extending this method to higher dimensions. Finally, we are continuing to test larger numbers of processes and datasets as well as perform tests where multiple CPUs share the same GPU resource. The results from these optimizations and tests will be reported at a later date.

References

[1] P. M. Kogge, S. Borkar, W. W. Carlson, W. J. Dally, M. Denneau, P. D. Franzon, S. W. Keckler, D. Klein, R. F. Lucas, S. Scott, A. E. Snavely, T. L. Sterling, R. S. Williams, K. A. Yelick, W. Harrod, D. P. Campbell, K. L. Hill, J. C. Hiller, S. Karp, M. Richards, and A. J. Scarpelli, "Exascale Study Group: Technology Challenges in Achieving Exascale Systems," DARPA, Tech. Rep., 2008.

[2] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM J. Matrix Analysis Applications*, vol. 32, no. 3, pp. 866–901, 2011.

[3] P. Swartztrauber, *Vectorizing the FFTs*. Acad. Press, 1982, pp. 51–83.

[4] M. Frigo and S. Johnson, "FFTW: An adaptive software architecture for the FFT," *ICASSP Conf. Proc.*, vol. 3, pp. 1381–1384, 1998.

[5] ——, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93(2), pp. 216–231, 2005.

[6] C. V. Loan, *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.

[7] P. Duhamel and H. Hollmann, "'split radix' FFT algorithm," *Electronics Letters*, vol. 20, no. 1, pp. 14 –16, 5 1984.

[8] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[9] R. Nishtala, P. Hargrove, D. Bonachea, and K. Yelick, "Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[10] P. T. P. Tang, J. Park, D. Kim, and V. Petrov, "A framework for low-communication 1-d FFT," in *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*, 2012.

[11] A. Edelman, P. McCorquodale, and S. Toledo, "The future fast Fourier transform?" in *PPSC*. SIAM, 1997.

[12] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, pp. 325–348, Aug. 1987.

[13] B. Cipra, "The best of the 20th century: Editors name top 10 algorithms," *SIAM News (Society for Industrial and Applied Mathematics)*, vol. 33, no. 4, p. 2, 2000.

[14] Committee on the Mathematical Sciences in 2025; Board on Mathematical Sciences And Their Applications; Division on Engineering and Physical Sciences; National Research Council, *Fueling Innovation and Discovery: The Mathematical Sciences in the 21st Century*. The National Academies Press, 2012.

[15] I. Lashuk, A. Chandramowlishwaran, M. H. Langston, T.-A. Nguyen, R. S. Sampath, A. Shringarpure, R. W. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, 2009.

[16] A. Dutt, M. Gu, and V. Rokhlin, "Fast algorithms for polynomial interpolation, integration, and differentiation," vol. 33, no. 5, pp. 1689–1711, Oct. 1996.

[17] A. Dutt and V. Rokhlin, "Fast Fourier transforms for nonequispaced data," *SIAM J. Sci. Comput.*, vol. 14, no. 6, pp. 1368–1393, Nov. 1993.

[18] R. Beatson and L. Greengard, "A short course on fast multipole methods," in *Wavelets, multilevel methods and elliptic PDEs*, M. Ainsworth *et al.*, Eds. Oxford University Press, 1997, pp. 1–37.

[19] M. H. Langston, L. Greengard, and D. Zorin, "A free-space adaptive FMM-based PDE solver in three dimensions," *Comm. in Applied Mathematics and Comp. Science*, vol. 6, no. 1, pp. 79–122, 2011.

[20] L. Greengard and W. Gropp, "A parallel version of the fast multipole method-invited talk," in *PPSC*, G. H. Rodrigue, Ed. SIAM, 1987, pp. 213–222.

[21] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical power consumption analysis and modeling for GPU-based computing," in *Proc. of ACM SOSP Workshop on Power Aware Comp. and Sys. (HotPower)*, 2009.

[22] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin, "R-Stream compiler," in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed. Springer, 2011, pp. 1756–1765.