Low Communication FMM-Accelerated FFT on GPUs

Cris Cecka
NVIDIA
Santa Clara, California 95050
ccecka@nvidia.com

ABSTRACT

Communication-avoiding algorithms have been a subject of growing interest in the last decade due to the growth of distributed memory systems and the disproportionate increase of computational throughput to communication bandwidth. For distributed 1D FFTs, communication costs quickly dominate execution time as all industry-standard implementations perform three all-to-all transpositions of the data. In this work, we reformulate an existing algorithm that employs the Fast Multipole Method to reduce the communication requirements to approximately a single all-to-all transpose. We present a detailed and clear implementation strategy that relies heavily on existing library primitives, demonstrate that our strategy achieves consistent speed-ups between 1.3x and 2.2x against cuFFTXT on up to eight NVIDIA Tesla P100 GPUs, and develop an accurate compute model to analyze the performance and dependencies of the algorithm.

KEYWORDS

FFT, FMM, GPU, Multi-GPU

ACM Reference format:
Cris Cecka. 2017. Low Communication FMM-Accelerated FFT on GPUs. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 11 pages.
https://doi.org/10.1145/3126908.3126919

1 INTRODUCTION

The cost of communication relative to the speed of computation on modern architectures has recast how algorithmic innovation is defined and performed, changing the metrics and goals of efficient algorithms [5]. Communication-avoiding and communication-reducing algorithms have demanded great interest in the past decade.

Of particular interest has been the Fast Fourier Transform (FFT), a staple of mathematical computing as it has been named one of the most important algorithms of the 20th century [3] (also on the list, the FMM). Many optimized, production-level implementations of the FFT exist across nearly all architectures, including FFTW [10], FFTPACK [22], cuFFT [19], and MKL DFTI [14]. Some of these libraries also support distributed implementations of the FFT, although renewed effort to reduce communication and energy consumption has spurred research into improving distributed implementations [2, 11, 18].

In addition, large amount of interest has been invested in the fast multipole method (FMM) [12, 16, 26], which is particularly suited for high performance by requiring local dense linear computations while remaining globally sparse and admitting hierarchical and parallelizable computation, reduced communication, and compressed representations of data and operators. Furthermore, compressed and dense algorithms of this type often harmoniously improve the energy-efficiency of the computations as well [17].

In this work, we revisit an algorithm to reduce the communication required in the Fast Fourier Transform (FFT) by up to 3x using FMMs and dense linear algebra [8, 15]. The original work recognized the potential communication savings, but no previous work has demonstrated these savings can be realized on existing architectures when compared to optimized FFT libraries. With the very high compute to communication ratio of NVIDIA's recently released DGX-1 node encasing eight NVIDIA Telsa P100 GPUs interconnected with NVLink, the benefits of the FMM-FFT can be realized with a careful reformulation of the algorithm that relies heavily on optimized primitives in cuBLAS and cuFFT. The novel contributions of this work include

- A clear and detailed implementation of the FMM-FFT in compact tensor notation and identification of stages that can be evaluated with optimized standard libraries.
- Generalizations of the FMM-FFT, specifically $P > G$ and $B > 2$, that allow it to perform efficiently using level-3 BLAS and optimized FFTs to achieve maximum performance.
- Results demonstrating consistent speed-ups up to 2.14x over the optimized and industry-standard cuFFTXT library on NVIDIA Telsa P100 GPUs.
- A performance model and parameter analysis demonstrating the implementation strategies achieve over 90% of peak practical performance.

2 RELATED WORK

Throughout this paper, we refer to Table 1 for notation.

For parallel distributed-memory in-order 1D FFTs, industry-standard implementations universally use variations of the original Cooley and Tukey algorithm [4], which rely on factorizations of the problem size, $N = MP$, and are detailed well in [25]. The common distributed memory implementations rely on three all-to-all communication steps (transpositions), which become the bottleneck for even moderate sized FFTs on modern architectures.

Edelman et al. [8] present an alternative factorization of the Fourier matrix that requires only a single all-to-all transposition, but also needs the FMM to retrieve $O(N \log N)$ performance. This FMM-FFT appears to be a generalization of a previous algorithm

SC17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5114-0/17/11…$15.00
https://doi.org/10.1145/3126908.3126919

by Dutt et al. [7] for nonequispaced FFTs, which can be interpreted as Edelman's formulation with $P = 1$. This work was revisited by Langston et al. [15] where the original Fortran code was updated with FFTW [10] and some GPU-accelerated computations were demonstrated. Langston et al. show the communication savings and provided some insight into the FMMs compute costs and trade-offs, but a clear and efficient implementation strategy and demonstration that the algorithm could be competitive to industry-standard libraries was lacking.

Tang et al. [24] developed an alternative algorithm based on convolution and oversampling which reduces the communication requirements to a single all-to-all transposition and a negligible halo exchange of the oversampled values. Park et al. [20] further analyze and present results for this method with Intel Xeon Phi coprocessors.

To our knowledge, the algorithms presented in [24] and [8] are the only in-order 1D FFTs that require a single all-to-all transposition while retaining the $O(N \log N)$ asymptotic work complexity. Both are approximate algorithms, however, and require careful bounding of the error and choice of parameters.

The FMM-FFT has several distinct advantages, some of which have been noticed by previous authors and some of which have been overlooked. Langston et al. notes the ability within the FMM-FFT to specify the error a priori regardless of the complexity or distribution of the input as well as the potential for parallelization of the compute stages within the FMM. Both [15, 24] note the perceived complexity of the FMM-FFT preventing its adoption and efficient implementation. Indeed, it remains unclear exactly how the the FMM-FFT was implemented in [8, 15] and all references to any standard linear algebra kernels are to matrix-vector products and GEMV, which are computationally suboptimal for the FMM-FFT.

3 FMM-FFT OVERVIEW

The Fourier matrix of size N is defined by

$$[\mathbf{F}_N]_{ij} = \omega_N^{ij} \tag{1}$$

where $\omega_N = \exp(-2\pi\imath/N)$. Direct application of the Fourier matrix would require $O(N^2)$ computations, but the famous FFT applies $\mathbf{F}_N$ exactly in $O(N \log N)$ computations via any number of equivalent matrix factorizations. In a distributed-memory setting, the radix-P splitting [25] is common and employs the factorizaton

$$\mathbf{F}_N = \mathbf{\Pi}_{M,P} \, (\mathbf{I}_M \otimes \mathbf{F}_P) \, \mathbf{\Pi}_{P,M} \, \mathbf{T}_{P,M} \, (\mathbf{I}_P \otimes \mathbf{F}_M) \, \mathbf{\Pi}_{M,P}$$

where $N = MP$, $[\mathbf{T}_{P,M}]_{ij} = \delta_{ij} \, \omega_N^{(i \bmod M) \cdot \lfloor i/M \rfloor}$ is a diagonal matrix of twiddle factors, and $\mathbf{\Pi}_{M,P}$ is a block-to-cyclic permutation which acts on $\widehat{\mathbf{e}}_i$, the ith column of the identity matrix $\mathbf{I}_N$, as

$$\mathbf{\Pi}_{M,P} \, \widehat{\mathbf{e}}_{p+mP} = \widehat{\mathbf{e}}_{m+pM} \qquad \begin{matrix} 0 \le p < P \\ 0 \le m < M \end{matrix}$$

The radix-P split FFT is then implemented in six steps:

(1) Transpose P-major to M-major.
(2) Compute P local FFTs of size M.
(3) Apply twiddle factors $\mathbf{T}$.
(4) Transpose M-major to P-major.
(5) Compute M local FFTs of size P.
(6) Transpose P-major to M-major.

Notation	Description
N	The size of the 1D FFT to perform.
M, P	$N = MP$. The FFT factors of N. There are P FMMs of size $M \times M$ to perform.
G	The number of distributed–memory processing elements.
L, B	$L \ge B \ge 2$. The leaf and base level of the FMM trees.
M_L	$M_L = M/2^L$. The number of points per leaf per FMM.
Q	The quadrature order of multipole and local expansions.
$\mathcal{M}^\ell, \mathcal{L}^\ell$	The multipole and local expansions at tree level ℓ.
$\mathcal{S}, \mathcal{T}$	The input (source) and output (target) of the FMMs.

Table 1: Table of notation.

These steps require three transpositions, which in a distributed setting result in three *all-to-all* communications between the processing elements.

The FMM-FFT [7, 8] instead factorizes the Fourier matrix as

$$\mathbf{F}_N = (\mathbf{I}_P \otimes \mathbf{F}_M) \, \mathbf{\Pi}_{M,P} \, (\mathbf{I}_M \otimes \mathbf{F}_P) \, \mathbf{\Pi}_{P,M} \, \mathbf{H}_{P,M} \, \mathbf{\Pi}_{M,P}$$

where

$$\mathbf{H}_{P,M} = \operatorname{diag}(\mathbf{I}_M, \mathbf{C}_1, \ldots, \mathbf{C}_{P-1})$$
$$[\mathbf{C}_p]_{mn} = \rho_p \left[\cot\left(\frac{\pi}{M} (n - m) + \frac{\pi}{N} p \right) + \imath \right]$$

and $\rho_p = \exp(-\imath\pi p/P) \sin(\pi p/P)/M$. Each $\mathbf{C}_p$ matrix is a *kernel matrix* and can be applied approximately with a periodic 1D FMM. Rather than perform all three transpositions in the factorization, we instead consider the matrix $\widehat{\mathbf{H}}_{M,P} = \mathbf{\Pi}_{P,M} \, \mathbf{H}_{P,M} \, \mathbf{\Pi}_{M,P}$, which can be interpreted as performing $P - 1$ distributed and interleaved FMMs. A distributed FMM requires communication, but much less communication than two transpositions. With this view, the distributed FMM-FFT can then be implemented as

(1) Compute $P - 1$ distributed FMMs of size $M \times M$.
(2) Compute M local FFTs of size P.
(3) Transpose P-major to M-major.
(4) Compute P local FFTs of size M.

Although it does not appear to be recognized in [8] or [15], we note that steps (2)–(4) are precisely a distributed 2D FFT of size $M \times P$, for which efficient implementations exist on many architectures. Thus, the FMM-FFT is essentially a two-stage computation,

$$\mathbf{F}_N = \mathbf{F}_{M,P} \, \widehat{\mathbf{H}}_{M,P} \tag{2}$$

where $\widehat{\mathbf{H}}_{M,P}$ is the interleaved $P - 1$ distributed periodic 1D FMMs of size M and $\mathbf{F}_{M,P}$ is the distributed $M \times P$ 2D FFT.

4 FMMS OF THE FMM-FFT

In this section, we detail our implementation strategies and optimizations of the FMMs within the FMM-FFT and refer to Table 1 throughout for notation. The FMMs to perform are one dimensional, periodic, and uniform with sources and targets located at the integers. Each FMM computes the action of one cotangent kernel matrix

$$[\widetilde{\mathbf{C}}_p]_{mn} = \cot\left(\frac{\pi}{M} (n - m) + \frac{\pi}{N} p \right).$$

The FMM decreases the computational cost of applying a $\mathbf{C}_p$ matrix from $O(M^2)$ to $O(M)$ for a fixed user-defined accuracy by partitioning the rows and columns hierarchically and applying a low-rank, implicit representation of the matrix. We use interpolative FMMs of the form found in [6, 9], but fuse operations so that every stage is a level-3 BLAS or BLAS-like dense computation, which we express in tensor notation.

4.1 Tensor Notation

In this work, we rely heavily on efficient means of expressing and computing dense linear algebra. To these means, we use tensor notation to compactly represent computations to be performed [21]. An array with multidimensional access,

$$A[i + j * \texttt{ldA<1>} + k * \texttt{ldA<2>} + \ell * \texttt{ldA<3>}],$$

will be written in *generalized column-major* form as

$$\mathcal{A}_{ijk\ell} \quad \begin{matrix} 0 \le i < \texttt{dimA<0>} \\ 0 \le j < \texttt{dimA<1>} \end{matrix} \quad \begin{matrix} 0 \le k < \texttt{dimA<2>} \\ -1 \le \ell < \texttt{dimA<3>}+1 \end{matrix}$$

with the dimensions of each tensor mode included where appropriate. For convenience, we allow the last index to also take on negative values or values larger than `dim`.

All tensors in this paper are represented *compactly*, where the leading dimension of the ith order is precisely the product of the dimensions of all previous orders,

$$\texttt{ldA<}i\texttt{>} = \prod_{k=0}^{i-1} \texttt{dimA<}k\texttt{>}.$$

Once the representation of a tensor has been established, we express tensor contractions in Einstein notation, but also include the iteration range of each mode involved in the contraction where appropriate.

$$C_{ipj} = \mathcal{A}_{ikp}\, \mathcal{B}_{kjp} \quad \begin{matrix} 0 \le i < I \\ 0 \le j < J \end{matrix} \quad \begin{matrix} 0 \le p < P \\ 0 \le k < K \end{matrix}$$

In the above, k is the "contraction index" as it appears in both tensors on the right, i and j are "row/col indices" as they appear once on the left and once on the right, and p is a "batch index" as it appears in all three tensors. We also allow iteration ranges to take on negative values or values larger than `dim`, which indicates overlapping iteration domains or iteration into halo regions.

4.2 FMM Representations

On each processing element, we store the input, the multipole expansions, the local expansions, and the output of the FMMs.

The input is presumed to be block-partitioned across processing elements and prescribed to be p-major by the factorization (2). Furthermore, we require a halo exchange of both end boxes in order to compute the S2T stage of Section 4.6.

$$\mathcal{S}_{pmb} \quad \begin{matrix} 0 \le p < P \\ 0 \le m < M_L \\ -1 \le b < 2^L/G+1 \end{matrix}$$

The output is also presumed to be block-partitioned across processing elements and prescribed to be p-major by the factorization (2), but does not require any halo space.

$$\mathcal{T}_{pmb} \quad \begin{matrix} 0 \le p < P \\ 0 \le m < M_L \\ 0 \le b < 2^L/G \end{matrix}$$

We choose to represent the multipole expansions as p-major. The $p = 0$ slice of the input does not participate in an FMM (since $\mathbf{C}_0 = \mathbf{I}_M$). At each non-base level of the tree, $\ell < B$, the multipole boxes are distributed across the processing elements and a halo exchange of two boxes on each end is required to compute the M2L stage of Section 4.7. At the base level of the tree, $\ell = B$, the M2L stage requires all boxes from all processing elements.

$$\mathcal{M}^{\ell}_{pqb} \quad \begin{matrix} 0 \le p < P-1 \\ 0 \le q < Q \\ -2 \le b < 2^{\ell}/G+2 \\ B < \ell \le L \end{matrix} \qquad \mathcal{M}^{B}_{pqb} \quad \begin{matrix} 0 \le p < P-1 \\ 0 \le q < Q \\ 0 \le b < 2^B \end{matrix}$$

We choose to represent the local expansions as p-major as well. The local expansion requires no halo space and is partitioned across the processing elements.

$$\mathcal{L}^{\ell}_{pqb} \quad \begin{matrix} 0 \le p < P-1 \\ 0 \le q < Q \\ 0 \le b < 2^{\ell}/G \end{matrix}$$

4.3 Basis Functions

Edelman et al. use an enhanced basis set that requires approximately two fewer basis functions to acheive a given level of accuracy [8]. This complicates the FMM by making the M2M and L2L operators depend on the level of the tree. Instead, we use Chebyshev basis functions to yield a simpler algorithm that requires less precomputation and storage overhead.

Define the Lagrange basis polynomials over the Chebyshev points of the first kind as

$$\ell_i(z) = \prod_{\substack{0 \le k < Q \\ k \ne i}} \frac{z - z_k}{z_i - z_k} \qquad \text{with} \qquad z_j = \cos\left(\frac{(2j+1)\pi}{2Q}\right).$$

4.4 S2M and L2T

The outgoing far-field representation (multipole expansion) at the leaf of the tree is constructed from the sources contained within that leaf via the S2M stage. At the leaf level, there are M_L sources per box per FMM. We map the sources of each leaf box to $[-1, 1]$ via

$$s_m = -1 + \frac{2m+1}{M_L}, \qquad 0 \le m < M_L.$$

The S2M operator,

$$S2M_{qm} = \ell_q(s_m) \quad \begin{matrix} 0 \le q < Q \\ 0 \le m < M_L \end{matrix},$$

is a matrix that maps input located at the source points s_m into the outgoing multipole coefficients. The same S2M matrix is used within each FMM and each box, so the S2M stage can be expressed as the tensor contraction

$$\mathcal{M}^{L}_{(p-1)qb} = S2M_{qm}\, \mathcal{S}_{pmb} \quad \begin{matrix} 1 \le p < P \\ 0 \le b < 2^L \end{matrix} \quad \begin{matrix} 0 \le q < Q \\ 0 \le m < M_L \end{matrix}$$

This computation can be performed with a single call to the BatchedGEMM primitive, which is a level-3 BLAS extension that is has optimized parallel implementations as of MKL 11.3β and cuBLAS 4.1 [21].

Similarly, the L2T stage maps incoming local expansion coefficients to output located at the target points. With the same mapping of the target points to the interval $[-1, 1]$, the L2T is the transpose operation of the S2M, but also accumulates with the result of the

S2T stage:

$$\mathcal{T}_{pmb} = L2T_{mq}\, \mathcal{L}^{L}_{(p-1)qb} + \mathcal{T}_{pmb} \qquad \begin{matrix} 1\le p<P & 0\le q<Q \\ 0\le b<2^L & 0\le m<M_L \end{matrix}$$

where

$$L2T_{mq} = S2M_{qm}.$$

This is also a tensor contraction that can be performed with a single call to BatchedGEMM.

4.5 M2M and L2L

The multipole coefficients of two sibling boxes are translated and combined into their parents' coefficients via the M2M stage. The representation of the parent box is also a polynomial expansion over $[-1, 1]$, implying the multipole coefficients of the two child boxes are located at the Chebyshev nodes scaled to $[-1, 0]$ and $[0, 1]$. Thus, we define a left (−) and a right (+) operator,

$$M2M^{\pm}_{qk} = \ell_q\left(\frac{z_k \pm 1}{2}\right)$$

and compute

$$\mathcal{M}^{\ell}_{pqb} = M2M^{-}_{qk}\, \mathcal{M}^{\ell+1}_{pk(2b+0)} + M2M^{+}_{qk}\, \mathcal{M}^{\ell+1}_{pk(2b+1)}$$

for each $\ell = L-1, \ldots, B$. Because the multipole coefficients are stored p-major, the above tensor contractions can be flattened into

$$\mathcal{M}^{\ell}_{pqb} = M2M_{qk}\, \mathcal{M}^{\ell+1}_{pk(2b)} \qquad \begin{matrix} 0\le p<P-1 & 0\le q<Q \\ 0\le b<2^{\ell} & 0\le k<2Q \end{matrix}$$

where

$$M2M = \begin{bmatrix} M2M^{-} & M2M^{+} \end{bmatrix}.$$

Each level's M2M computation can be performed with a single call to BatchedGEMM.

Similarly, the L2L stage translates the coefficients of a parent box into each of the child boxes, but will also accumulate this data with the result of the M2L stage. Performing the same scaled translation and flattening, the L2L stage is expressed as

$$\mathcal{L}^{\ell+1}_{pq(2b)} = L2L_{qk}\, \mathcal{L}^{\ell}_{pkb} + \mathcal{L}^{\ell+1}_{pq(2b)} \qquad \begin{matrix} 0\le p<P-1 & 0\le q<2Q \\ 0\le b<2^{\ell} & 0\le k<Q \end{matrix}$$

for each $\ell = B, \ldots, L-1$, where

$$L2L_{qk} = M2M_{kq}.$$

Each level's L2L computation can be performed with a single call to BatchedGEMM.

4.6 S2T

The S2T stage of a periodic 1D FMM applies a block-tridiagonal submatrix of each $\mathbf{C}_p$ matrix by direct multiplication. This is often expressed as a box interacting with every "neighbor" box, which can be written as the tensor contraction

$$\mathcal{T}_{pib} = S2T_{pijs}\, \mathcal{S}_{pj(b+s)} \qquad \begin{matrix} 0\le i,j<M_L & 0\le p<P \\ -1\le s\le 1 & 0\le b<2^L/G \end{matrix}$$

where we have defined the S2T tensor as elements of the $\mathbf{C}_p$ matrices

$$S2T_{pijs} = \begin{cases} \cot\left(\frac{\pi}{M}(j-i) + \frac{\pi}{2^L}s + \frac{\pi}{N}p\right) & p > 0 \\ \delta_{ij}\delta_{s0} & p = 0 \end{cases}$$

This can be simplified by flatting the s and j indices and recognizing the Toeplitz structure. That is, we redefine the S2T tensor as

$$S2T_{pk} = \begin{cases} \cot\left(\frac{\pi}{N}(p + Pk)\right) & p > 0 \\ \delta_{k0} & p = 0 \end{cases}$$

and perform the equivalent contraction

$$\mathcal{T}_{pib} = S2T_{p(j-i)}\, \mathcal{S}_{pjb} \qquad \begin{matrix} 0\le i<M_L & 0\le p<P \\ -M_L\le j<2M_L & 0\le b<2^L/G \end{matrix}$$

This can be interpreted as an "interleaved and overlapped convolution", where every P elements of $\mathcal{S}$ participate in up to three convolutions of length $3M_L$.

4.7 M2L

The M2L stage of the FMM applies dense matrices by direct multiplication to each box of the multilevel tree. Every box receives a contribution from three non-neighbor "cousin" boxes, which can be written as

$$\mathcal{L}^{\ell}_{pib} = M2L^{\ell}_{pijs}\, \mathcal{M}^{\ell}_{pj(b+s)} \qquad \begin{matrix} 0\le p<P-1 & 0\le b<2^{\ell} \\ 0\le i,j<Q & s=\{-2,2,3\}\ (b \text{ even}) \\ & s=\{-3,-2,2\}\ (b \text{ odd}) \end{matrix}$$

where

$$M2L^{\ell}_{pijs} = \cot\left(\frac{\pi}{2^{\ell}}\left(\frac{z_j}{2} - \frac{z_i}{2} + s\right) + \frac{\pi}{N}(p+1)\right).$$

At the base level, the contraction is performed with all non-neighbor boxes,

$$\mathcal{L}^{B}_{pib} = M2L^{B}_{pijs}\, \mathcal{M}^{B}_{pj(b+s)} \qquad \begin{matrix} 0\le p<P-1 & 0\le b<2^B \\ 0\le i,j<Q & 2\le s\le 2^B-2 \end{matrix},$$

where the indexing in $\mathcal{M}^B$ is interpreted cyclically.

Both of these operations are dense and can be implemented with high computational intensity. We employ the logical structuring strategies of [23]. For 1D FMMs this amounts to treating sibling boxes together, along with standard tiling strategies for dense computations to achieve high performance implementations of these custom operations.

Note that with $B = 2$, each box at the base level has only one non-neighbor box. The motivation for the cut-off base level, $B \ge 2$, is the ability to trade the continued replication of the local essential trees and the latencies of the communication and computations therein for a single all-to-all gather of the multipole expansions and a dense computation [13].

4.8 Reduction

The constant ι term in each $\mathbf{C}_p$ matrix induces a reduction across the input for each $p > 0$,

$$r_{p-1} = 1_{mb}\, \mathcal{S}_{pmb} \qquad \begin{matrix} 1\le p<P \\ 0\le m<M_L \\ 0\le b<2^L \end{matrix}$$

To compute r_p more efficiently, we recognize that every column of the $S2M$ and $M2M$ matrices sum to one by construction. Thus, these matrices preserve the sum of elements of vectors that they act on and

$$r_p = 1_{qb}\, \mathcal{M}^{B}_{(p-1)qb} \qquad \begin{matrix} 1\le p<P \\ 0\le q<Q \\ 0\le b<2^B \end{matrix}$$

This can be computed with a single GEMV on the compressed data at the base level of the tree.

4.9 Summary

Algorithm 1 summarizes the steps detailed in Section 4 for performing $P-1$ FMMs within the FMM-FFT, followed by the post-processing and the 2D FFT. Many of these steps have no dependencies and can be executed in parallel. For example, the S2M overlapped with the $\mathcal{S}$-halo communication, the S2T can be executed in parallel with any portion of the far-field computation and overlapped with the $\mathcal{M}$-communication stages, etc. On GPU, this parallelism is accomplished with CUDA streams and asynchronous communication.

Algorithm 1 The FMM-FFT via tensor contractions.

1: $\mathcal{M}^L_{(p-1)qb} = S2M_{qm}\,\mathcal{S}_{pmb}$ // S2M
2: SendRecv $\mathcal{S}$ halo. // COMM $\mathcal{S}$
3: $\mathcal{T}_{pib} = S2T_{p(j-i)}\,\mathcal{S}_{pjb}$ // S2T
4: **for** $\ell = L-1,\ldots,B$ **do**
5: $\quad\mathcal{M}^\ell_{pqb} = M2M_{qk}\,\mathcal{M}^{\ell+1}_{pk(2b)}$ // M2M
6: **for** $\ell = L,\ldots,B-1$ **do**
7: $\quad$SendRecv $\mathcal{M}^\ell$ halo. // COMM $\mathcal{M}^\ell$
8: $\quad\mathcal{L}^\ell_{pib} = M2L^\ell_{pijs}\,\mathcal{M}^\ell_{pj(b+s)}$ // M2L-ℓ
9: All-to-all gather $\mathcal{M}^B$. // COMM $\mathcal{M}^B$
10: $\mathcal{L}^B_{pib} = M2L^B_{pijs}\,\mathcal{M}^B_{pj(b+s)}$ // M2L-B
11: $r_p = 1_{qb}\,\mathcal{M}^B_{(p-1)qb}$ // REDUCE
12: **for** $\ell = B,\ldots,L-1$ **do**
13: $\quad\mathcal{L}^{\ell+1}_{pq(2b)} = L2L_{qk}\,\mathcal{L}^\ell_{pkb} + \mathcal{L}^{\ell+1}_{pq(2b)}$ // L2L
14: $\mathcal{T}_{pmb} = L2T_{mq}\,\mathcal{L}^L_{(p-1)qb} + \mathcal{T}_{pmb}$ // L2T
15: $\mathcal{T}_{pmb} = \rho_p(\mathcal{T}_{pmb} + \imath r_p)$ // POST
16: $\mathrm{F}_{M,P}\,\mathcal{T}$ // 2D FFT

An additional optimization notes that cuFFTXT provides customizable input and output callback functions [19] which can be used to abstract the load and store operations. This is used to fuse the post-processing stage (Line 15) and the 2D FFT (Line 16), preventing one extra round trip of $\mathcal{T}$ through global memory.

5 ANALYSIS

The FMM-FFT of Section 4 is reformulated so that every stage is implemented as a dense operation with high compute intensity. Thus, it should admit an accurate roofline model that we may use to predict the performance of the FMM-FFT.

In computing this model, many of the stages' floating point operation count and memory operation count involve sums over the levels of the tree. For convenience, we define

$$\sum_{\ell=B}^{L-1}\left\lceil\frac{2^\ell}{G}\right\rceil = \begin{cases} 2^L/G - 2^B/G & B > \log G \\ 2^L/G - B - 1 + \log G & B \le \log G \end{cases}$$
$$\equiv \frac{2^L}{G} - v(B,G) \equiv v(L,B,G).$$

with the assumption $L > \log G$.

5.1 Flops

Note that each FMM is real-valued and all operators of Section 4 are real-valued. When applied to complex input, the FMM stage requires only 2x the number of flops. Furthermore, with complex data layout as array-of-structs of the real and complex components, the p-major tensor data layout used in Section 4 is such that each real-complex matrix-matrix multiply can be flattened into a single real-real matrix-matrix multiply. Let C be 1 if the input is real and 2 if the input is complex. The the total flop count for each stage is

- **S2M, L2T:** $2CM_L2^L(P-1)Q/G$.
- **M2M, L2L:** $4C(2^L/G - v(B,G))(P-1)Q^2$.
- **S2T:** $6CM_L^2 2^L(P-1)/G$.
- **M2L-ℓ:** $6C(2^{L+1}/G - v(B+1,G))(P-1)Q^2$.
- **M2L-B:** $2C2^B(2^B-3)(P-1)Q^2/G$.
- **Reduce:** $C2^B(P-1)Q$.

Collecting terms and substituting $2^L = N/PM_L$,

$$C\left[20\frac{Q^2}{M_L} + 6M_L + 4Q\right]\left(1-\frac{1}{P}\right)\frac{N}{G}$$
$$+O\left(C(2^B(2^B-3)/G - v(B,G))(P-1)Q^2\right)$$

The first three terms agree precisely with Edelman's flop count [8] when $P = G$, $C = 2$, and $B = 2$, but demonstrates the weak dependence on P when $P > G$. The intuition is that by doubling P, the flops of each tensor contraction doubles but M halves and each FMM requires one less level in the tree. The constants in front of the last three terms depends on precisely how the top of the tree–when the number of boxes is less than the number of processes–is handled. We include $B \ge 2$ to potentially trade latency/communication dominated steps at the very top of the tree for a computation dominated M2L between all pairs of non-neighbor boxes. See [13] for more details on local essential trees in FMMs.

5.2 Communication

We agree with the communication analysis presented in [8, 15]. Each process sends

- $\mathcal{S}$: $2C(P-1)M_L$.
- $\mathcal{M}^\ell$: $4C(L-B)(P-1)Q$.
- $\mathcal{M}^B$: $2^BC(P-1)Q$.

This is extremely small compared to the number of flops performed. The cost and latency of the communication can be reliably hidden by overlapping with the FMMs' computation at the scales considered in this work.

5.3 Mops

The number of memory reads and writes required by Algorithm 1 in each stage is

- **S2M+L2T:** $2QM_L + 3C(P-1)M_L2^L/G + 2C(P-1)Q2^L/G$.
- **M2M+L2L:** $4Q^2 + (2+1+4+1)C(P-1)Q(2^L/G - v(B,G))$.
- **S2T:** $4M_L(P-1) + (2^L/G + 2 + 2^L/G)CM_L(P-1)$.
- **M2L-ℓ:** $4(P-1)Q^2(L-B) + 2v(L+1,B+1,G)C(P-1)Q$.
- **M2L-B:** $(2^B-3)(P-1)Q^2 + (2^B + 2^B/G)C(P-1)Q$.
- **Reduce:** $C(P-1) + C2^B(P-1)Q$.

Collecting and keeping dominant terms, the total memory operations is bounded below by

$$2QM_L + 4Q^2 + 4PM_L + PQ^2(4\log\frac{N}{M_L P} - 4B + 2^B - 3) + C\left(5 + 14\frac{Q}{M_L}\right)\left(1 - \frac{1}{P}\right)\frac{N}{G} + O(C(2^B + 2^B/G - v(B,G))(P-1)Q)$$

where the first line of terms come from the FMM operators, the second line is the reads/write of the input and output of the multipole and local expansions, and the last term depends on how the top of the tree is handled.

This count could be further lowered by additional fusion of the FMM steps. For example, the M2L and L2L stages could be fused to prevent 1 read and 1 write ($2Q/M_L$ of the second line) of the $\mathcal{L}$ data. This, however, requires more modular, composable BLAS primitives capable of performance on par with cuBLAS.

Note that the PM_L and PQ^2 terms come from the S2T and M2L operators. As these stages require custom BLAS-like kernels, we choose to compute the entries of the S2T and M2L operators on-the-fly rather than reading them from memory. When these operators are not considered in the mop count, the computational intensity of the FMMs actually increases with P, providing a computation-memory trade-off.

5.4 Practical Roofline

The S2M, M2M, L2L, and L2T stages can all be expressed as the level-3 BLAS primitive BatchedGEMM available in cuBLAS 8.0 which achieves very high efficiency on the P100 GPU [21]. In Figure 1, the performance of a pure GEMM of size $N^2 \times N \times N$ is compared with the performance of a BatchedGEMM that computes N matrix-matrix multiplies of size $N \times N \times N$.

From the GEMM performance, we define the practical architecture parameters

$$\gamma_{f,K40} = 2.8 \text{ TFlop/sec} \qquad \gamma_{f,P100} = 10 \text{ TFlop/sec}$$
$$\gamma_{d,K40} = 1.2 \text{ TFlop/sec} \qquad \gamma_{d,P100} = 5 \text{ TFlops/sec}$$
$$\beta_{K40} = 100 \text{ GByte/sec} \qquad \beta_{P100} = 360 \text{ GBytes/sec}$$

and the model minimum wall-time for a computation as

$$T = W/\min(\gamma, \beta * W/D) \tag{3}$$

where T is the compute time in seconds, W is the total compute operation count in TFlops, and D is the total memory operation count in GBytes.

6 FMM-FFT RESULTS

The experimental setup consists of two systems. Two K40c GPUs with achieved $\overline{\beta}_{K40} = 13.2$ GB/s P2P interconnect via PCIe and eight P100 GPUs with achieved $\overline{\beta}_{P100} = 36$ GB/s P2P interconnect via NVLink.

In Figure 2 (top), the profile of the 1D FFT for $N = 2^{27} = 1.34\times10^8$ shows it is severely communication bound–each yellow bar is asynchronous P2P communication–due to the three transposition steps despite nearly perfect overlap of the communication and computation. In contrast, the profile of the FMM-FFT in Figure 2 (bottom) shows a large amount of computation–all the non-yellow kernels–followed by a 2D FFT which also overlaps significant portions of the communication and computation. The computation in the FMM portion of the FMM-FFT is parallelized via CUDA streams and also overlaps all of the FMM communication.

In total, Figure 2 shows 255 FMMs of size $524k \times 524k$ are computed in 32ms with 35 kernel launches:

- **S2M:** 1 BatchedGEMM.
- **M2M:** 10 BatchedGEMMs, one for each level.
- **S2T:** 1 custom kernel launched 1 time.
- **M2L:** 1 custom kernel launched 11 times.
- **Reduce:** 1 GEMV.
- **L2L:** 10 BatchedGEMMs, one for each level.
- **L2T:** 1 BatchedGEMM.

For large N, the data transpositions dominate the performance of the 1D and 2D FFTs. The FMM-FFT saves two transpositions and the theoretical cross-over point can be approximated in terms of the architecture parameters

$$\frac{\overline{\beta}}{\min(\gamma, \beta \cdot W/D)} = \frac{N}{W}$$

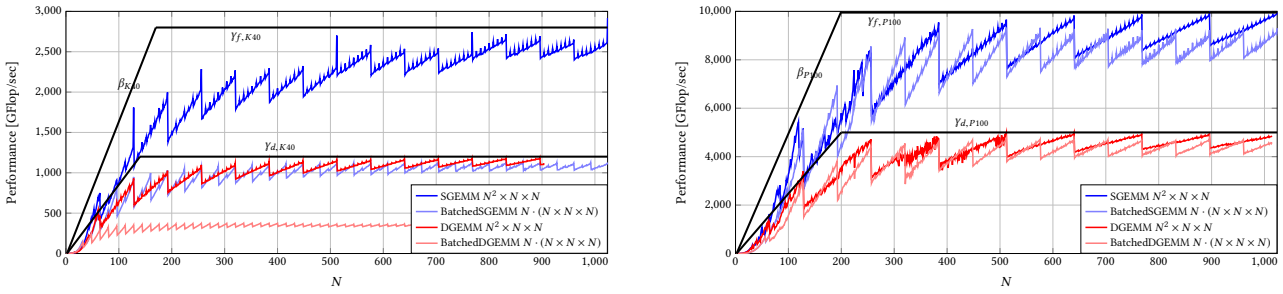
For large N on P100, this ratio is computed to be approximately 0.031 byte/flop, which agrees well with 0.036 computed in [8]. With the architecture parameters from Section 5.4, the communication to flop ratio of the K40c system is approximately 0.0012 byte/flop and the P100 system is approximately 0.0009 byte/flop. However, the model intensity for the FMM-FFT in this regime is only 7.8 flops/byte in double precision, so the roofline peak performance is only 2.7 TFlops in double-precision on P100. That is, the FMM stage remains slightly memory bound and the true predictor of success is not the communication to computation ratio but rather the communication to memory bandwidth ratio and the expected computational intensity. Regardless, the memory bandwidth to communication ratios are increasing in addition to computation to communication ratios, so algorithms such as the FMM-FFT remain likely to become increasingly important.

6.1 Performance

The FMM-FFT is executed in single, double, single-complex, and double-complex precision on each system and for each set of admissible parameters. Figure 3 reports the performance of the fastest FMM-FFT normalized to the performance of the 1D FFT from cuFFTXT on the same architecture and input. All reported results achieve less than 4×10^{-7} relative ℓ_2 error in single-complex precision and 2×10^{-14} relative ℓ_2 error in double-complex precision.

On 2xK40c GPUs, the FMM-FFT is only marginally faster than the 1D FFT from cuFFTXT. This is due to the smaller architecture parameter ratios and the performance deficits of BatchedGEMM within cuBLAS 8.0 documented in Section 5.4. On 2xP100, the FMM-FFT realizes significant gains of 1.3x over the 1D FFT for large N in both single and double-precision. On 8xP100, the FMM computation is scaled nearly perfectly, but the FFT communication scales more poorly, allowing the FMM-FFT to achieve approximately 2.1x over the 1D FFT.

Interpreting the FMMs as a method for converting between a 2D FFT and a 1D FFT, the black bar indicates the budget of time that the FMMs have to work within. In optimized FFT libraries, distributed



(a) K40c GPU (b) P100 GPU

Figure 1: The performance of cuBLAS GEMM and BatchedGEMM on the K40c and P100 GPUs. The FMM-FFT relies heavily on the performance of BatchedGEMM and the roofline model predicts the maximum performance of a computation from its computational intensity. The roofline architecture parameters defined in Section 5.4 are overlaid in each plot.

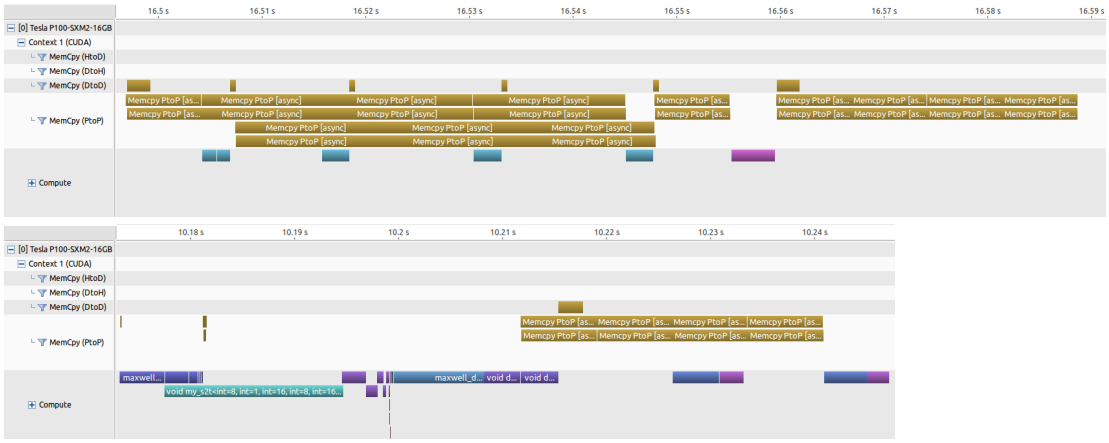


Figure 2: Performance profiles of the in-order, double-complex 1D FFT from cuFFTXT (top) and the FMM-FFT (bottom) with $N = 2^{27}$ on 2xP100 GPUs connected via NVLINK. The yellow is asynchronous communication and all other kernels are compute. The FMM-FFT uses parameters $P = 256, M_L = 64, B = 3, Q = 16$. Each profile shows only one of the two GPUs for brevity.

2D FFTs often achieve nearly 3x performance of distributed 1D FFTs precisely due to the avoidance of two of the three transpositions [10, 11].

The red bar indicates the maximum practical speed-up over the 1D FFT that can be achieved with Algorithm 1 predicted by the roofline model of Section 5.4. For $N \lesssim 2^{21}$, the performance of the distributed 1D and 2D FFTs in cuFFTXT is latency and synchronization bound, which also causes the roofline model to fail as latency is not taken into account. Regardless, the FMM-FFT still outperforms the 1D FFT in this regime as it has fewer synchronizations and fewer kernel launches for small N. For large N, the roofline model becomes an effective predictor of performance and we use it in the next section to provide a more complete view of the FMM-FFT as a function of each of its parameters.

6.2 Component Efficiency

Figure 4 shows the breakdown of the time spent in each kernel of the double-complex FMM stage on 2xP100. When N is small and performance is dominated by latency, the fastest FMM-FFT also minimizes the total number of kernel launches by keeping $L = B$, which requires the M2L-B and S2T stages to perform the majority of the computation. When N is large, the M2L-B stage is negligible and the time is dominated by BatchedGEMM and the S2T stage. This is a significant divergence from most FMM studies where the M2L and S2T stages are the most expensive and balancing their costs is primary.

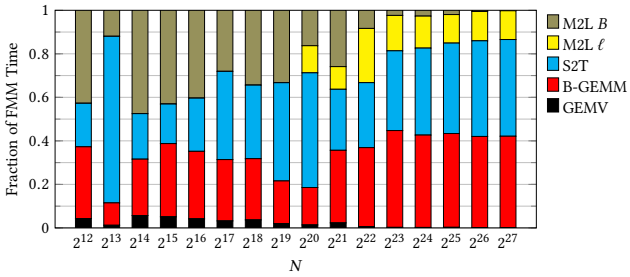


Figure 4: The fraction of time spent in each kernel of the double-complex FMM stage on 2xP100. The BatchedGEMM and S2T kernels dominate performance for large N.

To demonstrate the benefits of our implementation strategy, Figure 5 plots the efficiency of each stage of the FMM, the entire

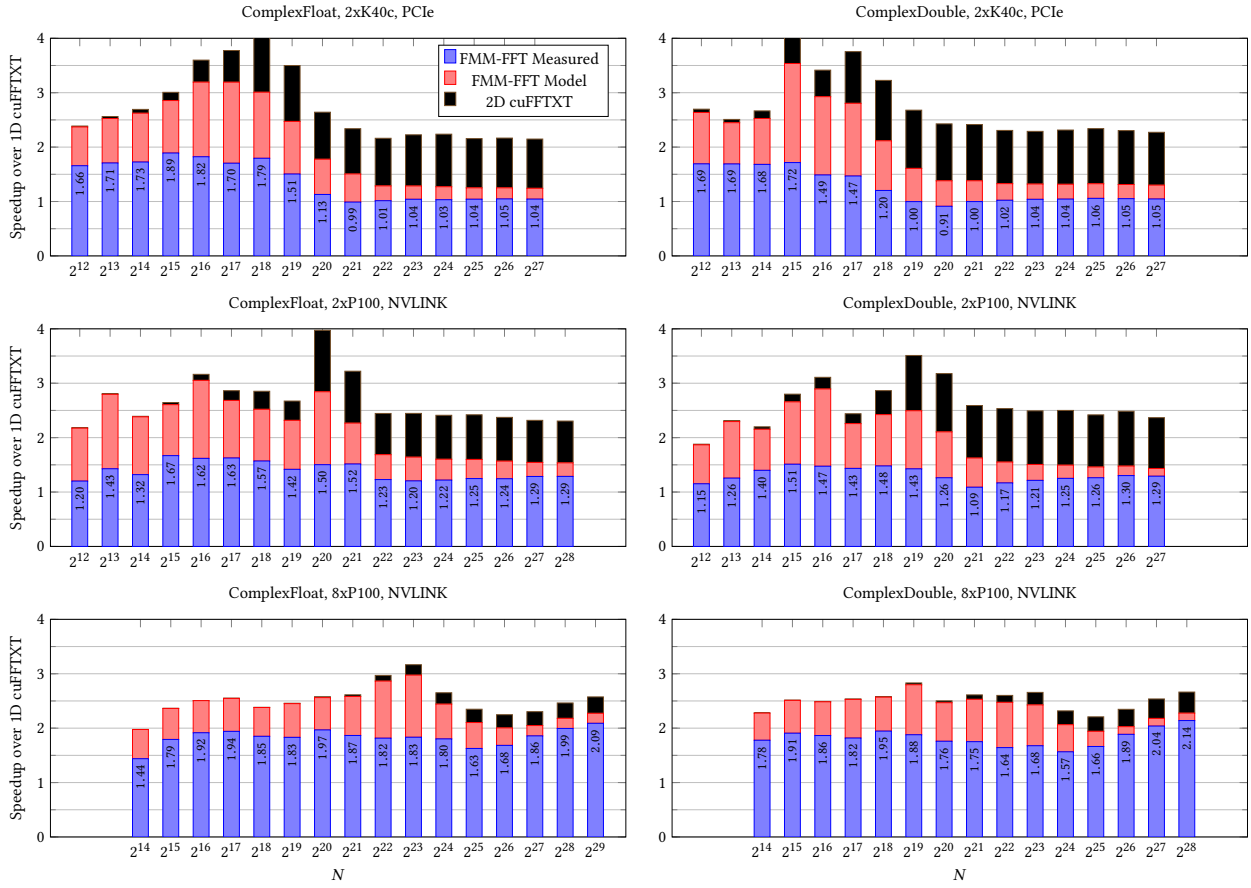


Figure 3: Speedup of the FMM-FFT over the 1D FFT from cuFFTXT, NVIDIA's multi-GPU FFT library. For each N, only the fastest FMM-FFT found by searching the parameter space is reported along with its peak practical performance from the roofline model and 2D FFT stage.

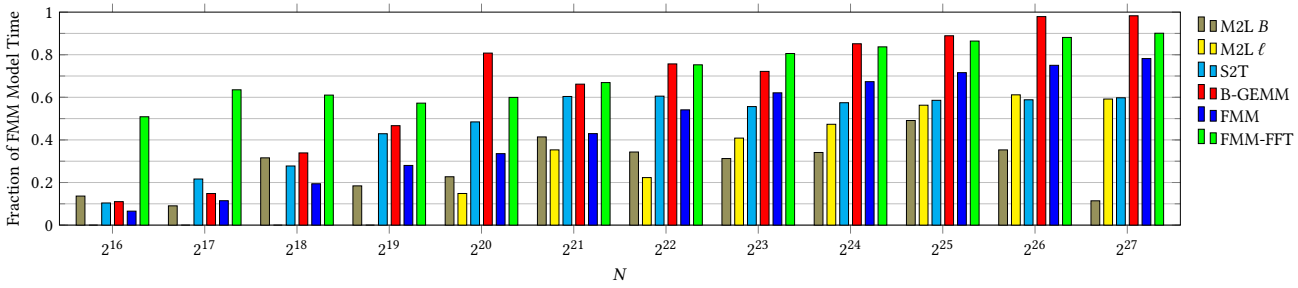


Figure 5: The efficiency of each stage of the FMM-FFT as the achieved fraction of the roofline model performance. For the FMM-FFT efficiency, we assume the measured 2D FFT implementation is 100% efficient, indicating the gains that can be achieved by optimizing the FMM stage only.

FMM stage, and the entire FMM-FFT. The efficiency is computed as the ratio of the roofline minimum wall-time, Equation (3), for that stage to the measured time spent in that stage. The M2L-B stage appears to consistently be the most inefficient but negligible for large N, while the BatchedGEMM stage is the most efficient and critical for large N. The implementations of the M2L-ℓ and S2T appear to achieve approximately 60% of their peak performance, which is about expected for dense linear kernels implemented in CUDA rather than assembly [1].

For the full FMM-FFT efficiency, we've assume the 2D FFT is 100% efficient in order to highlight the gains that could be achieved by further optimizing the FMM stage. The FMM-FFT is achieving approximately 90% of its peak performance and optimizing the FMM stage is a practice of significantly diminishing returns. Instead, further optimizations should be performed on the 2D FFT itself.

Additional kernel fusion optimizations within Algorithm 1 that reduce the total memory operations are likely to have the highest impact on the performance of the FMM stage rather than further optimizing the S2T and M2L stages.

6.3 Parameter Dependence

In the following sections, we investigate each parameter of the FMM-FFT independently using the roofline model constructed in Section 5 and measured performance.

6.3.1 M_L Dependence. The performance of the FMM-FFT strongly depends on M_L, the number of points per leaf box per FMM, by controlling the amount of computation that the near-field and far-field stages each perform. The computational cost of the S2T stage for constant N grows with $M_L^2 2^L = M_L N/P$ while the computational cost of the far-field decreases with $2^L = N/M_L P$ as the depth of the tree decreases.

Edelman et al. [8] and Langston et al. [15] count the number of floating point operations in the near-field and far-field as a function of M_L to minimize the computation performed and balance the costs. However, the number of operations performed is not directly proportional to performance in this case. Increasing M_L causes the total operation count of the S2T stage to increase but also increases its computational intensity. The total operation count of the far-field stages decrease, but the computational intensity remains nearly the same (only the S2M and L2T gain a small amount of intensity).

In Figure 6, we plot the total operation count of the FMM stage as a function of M_L for $N = 1.34 \times 10^8$, $P = 256$, $B = 3$, $G = 2$ and compare with the performance predicted by the roofline model. As shown, the optimal M_L for performance is higher than would be predicted by the flop count and measured performance agrees with the model. Most of the large-N cases in this paper use $M_L = 64$ rather than $M_L = 32$ as in [8, 15].

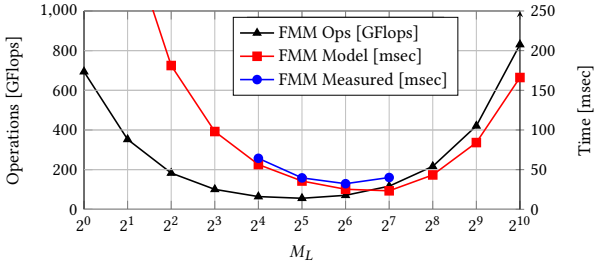


Figure 6: Dependence on M_L of the performance of the FMM stage with $N = 2^{27}, P = 256, B = 3, G = 2$.

6.3.2 P Dependence. The performance of the FMM-FFT depends more weakly on P, the number of FMMs to be performed. Increasing P with constant N does not significantly change the total operation count except at the very top of the tree. Furthermore, increasing P does not significantly change the computational intensity of the FMM stage either.

The largest influence that P has on performance comes from the FFT and secondarily from the BatchedGEMM. 2D FFTs are often optimized for square cases [10, 11, 19] where the computation and communication can be more effectively overlapped. Figure 7 demonstrates this by plotting the flop count, model time, and measured cuFFTXT 2D FFT time as a function of P for $N = 1.34 \times 10^8$ with $M_L = 64$, $B = 3$, $G = 2$. For large aspect ratios, the 2D FFT performance is approximately 3x slower than for more square cases. In fact, cuFFTXT rejects 2D FFTs with one dimension less than 32.

The measured FMM time does not include the post-processing or 2D FFT stages. As expected, the performance is stable as P increases and agrees with model performance. The small performance degradation when $P = 32$ is primarily due to the resulting small GEMM sizes–the number of rows in each BatchedGEMM is only 62 in that case.

Thus, even though very small P is favored by the analysis, moderate or large P is favored in practice when using tuned BLAS and 2D FFT libraries.

6.3.3 B Dependence. Despite the analysis, for large N the dependence of performance on B is weaker than might be expected. In Figure 8, we plot the total operation count and the performance predicted by the roofline model as a function of B for $N = 1.34 \times 10^8$ with $P = 256$, $M_L = 64$, and $G = 2$. Only for $B \geq 11$ does the increased floating point operations at the base level begin to affect the performance.

Therefore, $B > 2$ can be used to effectively combat local essential tree replication and latency for small and moderate numbers of processing elements without negatively impacting performance.

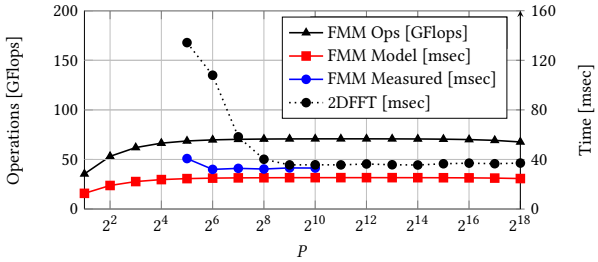


Figure 7: Dependence on P of the performance of the FMM stage with $N = 2^{27}, M_L = 64, B = 3, G = 2$. Small P degrades the performance of the 2D FFT as well as the BatchedGEMM.

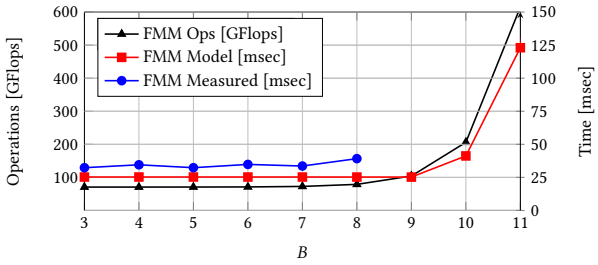


Figure 8: Dependence on B of the performance of the FMM stage with $N = 2^{27}, P = 256, M_L = 64, G = 2$.

6.3.4 Q Dependence. The Q parameter controls the expansion order of the FMM stage. Figure 9 (top) shows the relatively weak dependence of the flop count and model performance on Q. We do not show measured performance for the FMMs here because the

M2L kernel was tuned statically with $Q = 16$ for double-precision and $Q = 8$ for single-precision.

Figure 9 (bottom) demonstrates the achieved accuracy of the full double-complex FMM-FFT as a function of Q as compared to the 1D FFT from cuFFTXT in the relative ℓ_2 norm with each component generated uniformly in $[-1, 1]$. We observe the same odd-even behavior as Edelman et al. [8], but report an extra digit of accuracy, likely due to Edelman reporting the maximum error over all FMMs rather than the error of the full FMM-FFT. We find that accuracy is not improved for increasing Q above 18 due to machine precision and roundoff error.

FFTs that produce less accurate results are then potentially faster by 1.5x. Whether this can be realized practically–whether the efficiency of the BatchedGEMMs and M2L holds in this regime–remains to be seen.

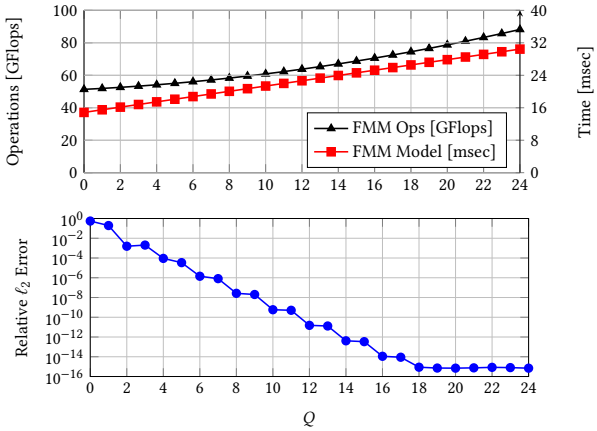


Figure 9: (Top) Dependence on Q of the performance of the FMMs with $N = 2^{28}, P = 128, M_L = 64, B = 3, G = 2$. (Bottom) Dependence on Q of the accuracy of the FMM-FFT.

7 CONCLUSION

We have presented a reformulation of the low-communication FMM-accelerated in-order 1D FFT from [8], detailed a clear and efficient algorithm to compute it, and demonstrated its success on 2xK40c and up to 8xP100 GPUs. To our knowledge, this is the first implementation that consistently outperforms a vendor-provided and industry-standard FFT library. To accomplish this, each stage of the FMM is presented as a dense tensor contraction to be computed with high intensity kernels. All but two of the stages of the FMM-FFT can be computed with existing optimized linear algebra primitives. Due to the reliance on dense computations, we show a roofline model accurately predicts the performance of the FMM-FFT, which we use to analyze performance with respect to each parameter. We measure up to 2.14x performance increase over cuFFTXT on 8xP100 GPUs. Because the implementation strategy in this work relies heavily on standard primitives, the algorithm is highly portable across architectures.

Further optimizations include exploiting additional symmetries of the operators *M2L*, *S2T*, *M2M*, and *S2M* to further reduce memory requirements and floating point operations, kernel fusion to reduce memory operations, and the use of more highly optimized 2D FFTs.

The results presented in this paper are all performed on a single node with GPUs connected by PCIe or NVLINK. Extending the results to multiple nodes is necessary and the reliance on standard primitives makes this much easier. In addition, the performance on multiple nodes is very likely to improve relative performance and energy efficiency due to higher internode communication costs.

REFERENCES

[1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. *Performance, Design, and Autotuning of Batched GEMM for GPUs.* Springer International Publishing, Cham, 21–38. https://doi.org/10.1007/978-3-319-41321-1_2

[2] Yifeng Chen, Xiang Cui, and Hong Mei. 2010. Large-scale FFT on GPU Clusters. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*. ACM, New York, NY, USA, 315–324. https://doi.org/10.1145/1810085.1810128

[3] B. A. Cipra. 2000. The Best of the 20th Century: Editors Name Top 10 Algorithms. *SIAM News* 33 (2000).

[4] James W. Cooley and John W. Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* 19, 90 (1965), 297–301. http://www.jstor.org/stable/2003354

[5] J Dongarra, J Hittinger, J Bell, L Chacon, R Falgout, M Heroux, P Hovland, E Ng, C Webster, and S Wild. 2014. *Applied Mathematics Research for Exascale Computing.* https://doi.org/10.2172/1149042

[6] A. Dutt, M. Gu, and V. Rokhlin. 1996. Fast Algorithms for Polynomial Interpolation, Integration, and Differentiation. *SIAM J. Numer. Anal.* 33, 5 (1996), 1689–1711. https://doi.org/10.1137/0733082 arXiv:http://dx.doi.org/10.1137/0733082

[7] A. Dutt and V. Rokhlin. 1995. Fast Fourier Transforms for Nonequispaced Data, II. *Applied and Computational Harmonic Analysis* 2, 1 (1995), 85 – 100. https://doi.org/10.1006/acha.1995.1007

[8] Alan Edelman, Peter McCorquodale, and Sivan Toledo. 1998. The Future Fast Fourier Transform? *SIAM Journal on Scientific Computing* 20, 3 (1998), 1094–1114. https://doi.org/10.1137/S1064827597316266 arXiv:http://dx.doi.org/10.1137/S1064827597316266

[9] William Fong and Eric Darve. 2009. The black-box fast multipole method. *J. Comput. Phys.* 228, 23 (2009), 8712 – 8725. https://doi.org/10.1016/j.jcp.2009.08.031

[10] M. Frigo and S. G. Johnson. 1998. FFTW: an adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, Vol. 3. 1381–1384 vol.3. https://doi.org/10.1109/ICASSP.1998.681704

[11] Amir Gholami, Judith Hill, Dhairya Malhotra, and George Biros. 2015. AccFFT: A library for distributed-memory FFT on CPU and GPU architectures. *arXiv preprint arXiv:1506.07933* (2015).

[12] Q. Hu, N. A. Gumerov, and R. Duraiswami. 2012. Scalable Distributed Fast Multipole Methods. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*. 270–279. https://doi.org/10.1109/HPCC.2012.44

[13] Huda Ibeid, Rio Yokota, and David Keyes. 2016. A performance model for the communication in fast multipole methods on high-performance computing platforms. *The International Journal of High Performance Computing Applications* 30, 4 (2016), 423–437. https://doi.org/10.1177/1094342016634819 arXiv:http://dx.doi.org/10.1177/1094342016634819

[14] Intel. 2017. Intel MKL DFTI Library. (2017).

[15] M. H. Langston, M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. 2013. Re-Introduction of communication-avoiding FMM-accelerated FFTs with GPU acceleration. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. 1–6. https://doi.org/10.1109/HPEC.2013.6670352

[16] Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros. 2012. A Massively Parallel Adaptive Fast Multipole Method on Heterogeneous Architectures. *Commun. ACM* 55, 5 (May 2012), 101–109. https://doi.org/10.1145/2160718.2160740

[17] Hatem Ltaief, Piotr Luszczek, and Jack Dongarra. 2012. Profiling high performance dense linear algebra algorithms on multicore architectures for power and energy efficiency. *Computer Science - Research and Development* 27, 4 (2012), 277–287. https://doi.org/10.1007/s00450-011-0191-z

[18] Lingchuan Meng, Jeremy Johnson, Franz Franchetti, Yevgen Voronenko, Marc Moreno Maza, and Yuzhen Xie. 2010. Spiral-Generated Modular FFT Algorithms. In *Parallel Symbolic Computation (PASCO)*. 169–170.

[19] Nvidia. 2017. CUDA cuFFT Library. (2017).

[20] Jongsoo Park, Ganesh Bikshandi, Karthikeyan Vaidyanathan, Ping Tak Peter Tang, Pradeep Dubey, and Daehyun Kim. 2013. Tera-scale 1D FFT with Low-communication Algorithm and Intel Xeon Phi Coprocessors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 34, 12 pages. https://doi.org/10.1145/2503210.2503242

[21] Yang Shi, U.N. Niranjan, Animashree Anandkumar, and Cris Cecka. 2016. Tensor Contractions with Extended BLAS Kernels on CPU and GPU. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 193–202. https://doi.org/10.1109/HiPC.2016.031

[22] Paul N. Swarztrauber. 1984. FFT Algorithms for Vector Computers. *Parallel Comput.* 1, 1 (Aug. 1984), 45–63. https://doi.org/10.1016/S0167-8191(84)90413-7

[23] Toru Takahashi, Cris Cecka, William Fong, and Eric Darve. 2012. Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units. *Internat. J. Numer. Methods Engrg.* 89, 1 (2012), 105–133. https://doi.org/10.1002/nme.3240

[24] P. T. P. Tang, J. Park, D. Kim, and V. Petrov. 2012. A framework for low-communication 1-D FFT. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. 1–12. https://doi.org/10.1109/SC.2012.5

[25] C. Van Loan. 1992. *Computational Frameworks for the Fast Fourier Transform.* Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611970999 arXiv:http://epubs.siam.org/doi/pdf/10.1137/1.9781611970999

[26] Rio Yokota and Lorena A Barba. 2012. A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems. *The International Journal of High Performance Computing Applications* 26, 4 (2012), 337–346. https://doi.org/10.1177/1094342011429952 arXiv:http://dx.doi.org/10.1177/1094342011429952