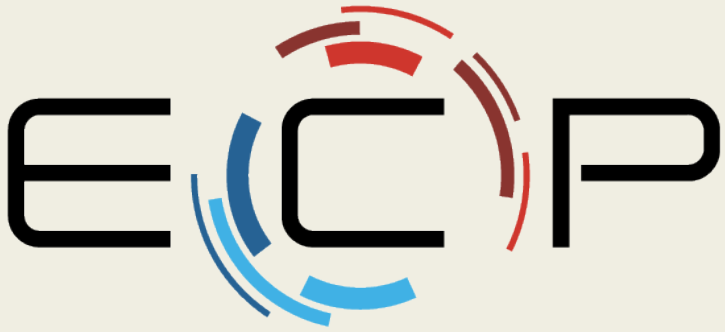

EXASCALE COMPUTING PROJECT

# ECP Milestone Report

## FFT-ECP API and High-Performance Library Prototype for 2-D and 3-D FFTs on Large-Scale Heterogeneous Systems with GPUs

WBS 2.3.3.13, Milestone FFT-ECP STML13-27

Stanimire Tomov
Alan Ayala
Azzam Haidar[1]
Jack Dongarra

Innovative Computing Laboratory, University of Tennessee

January 31, 2020

[1]The contribution of this author was done during the author's employment at the Innovative Computing Laboratory

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

| Revision | Notes |
|---|---|
| 01-2020 | first publication |

```
@techreport{thasd2020ECPFFT,
  author={Tomov, Stanimire and Ayala, Alan and Haidar, Azzam and Dongarra, Jack},
  title={{FFT-ECP API and high-performance library prototype for 2-D and 3-D FFTs on
          large-scale heterogeneous systems with GPUs}},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2020},
  month={January},
  type={ECP WBS 2.3.3.13 Milestone Report},
  number={FFT-ECP STML13-27},
  note={revision 01-2020}
}
```

# Contents

# List of Figures

# List of Tables

# CHAPTER 1

# Executive Summary

The goal of this milestone was the development of API and high-performance library prototype for 2-D and 3-D FFTs on large-scale heterogeneous systems with GPUs.

In this milestone we defined consistent FFT-ECP APIs for FFTs on Exascale systems that are suitable to ECP applications. We developed MPI communication optimizations to support CPU-based FFTs and GPU-accelerated FFTs using for CUDA-aware MPI. Specifically, this milestone delivered on the following sub-tasks:

- Study FFT use in applications;
- Generalize FFT APIs to fit application use;
- Add to FFT-ECP the new interfaces;
- Develop a benchmarking framework for MPI FFT communications;
- Develop MPI based optimizations for Summit.

The artifacts delivered include the performance optimizations, new features added to the solvers, auxiliary wrappers for use in applications of interest, and a tuned FFT-ECP software, freely available on the FFT-ECP's Git repository hosted on Bitbucket, https://bitbucket.org/icl/heffte/, under the name of Highly Efficient FFTs for Exascale (heFFTe). We released **heFFTe version 0.2**.

See also the FFT-ECP website, http://icl.utk.edu/fft/ for more details on the FFT-ECP project.

# CHAPTER 2

# Background

Considered one of the top 10 algorithms of the 20th century, the Fast Fourier transform (FFT) is widely used by applications in science and engineering. Such is the case of applications targeting exascale, e.g. LAMMPS (EXAALT-ECP) [14], and diverse software ranging from particle applications [20] and molecular dynamics, e.g. HACC [7], to applications in machine learning, e.g., [16]. For all these applications, it is critical to have access to a heterogeneous, fast and scalable parallel FFT library, with an implementation that can take advantage of novel hardware components and efficiently use them.

Highly efficient implementations to compute FFT on a single node have been developed for a long time. One of the most widely used libraries is FFTW [10], which has been tuned to optimally perform in several architectures. Vendor libraries for this purpose have also been highly optimized, e.g., as in the case of MKL (Intel) [13], ESSL (IBM) [8], clFFT (AMD) [1], and CUFFT (NVIDIA) [17]. Novel libraries are also being developed to further optimize single node FFT computation, e.g., FFTX [9] and Spiral [21]. Most of the previous libraries have been extended to distributed memory versions, some by the original developers, and others by different authors.

FFT-ECP leverages existing FFT capabilities by design, such as third-party 1-D FFTs from vendors or open-source libraries. This is also the approach in the `SWFFT` [22] and `FFTMPI` [19] FFT libraries, which are currently used in the HACC and the LAMMPS ECP application projects, respectively. `FFTMPI` and `SWFFT` have very good weak and strong scalability on CPU-based systems. The ECP-FFT project for this period developed a number of new features, extended APIs, and added communication and scalability optimizations for 2-D and 3-D FFTs. The supported features cover the `FFTMPI` and `SWFFT` functionalities for 2-D and 3-D FFTs on GPU-accelerated Exascale platforms. The software is released under the **heFFTe version 0.2** library, freely available on the FFT-ECP's Git repository hosted on Bitbucket, https://bitbucket.org/icl/heffte/.

# CHAPTER 3

# Optimizing FFT communications on GPUs

Even though the fast development of GPUs has enabled great speedups on local/single GPU computations, the cost of communication between CPUs and/or GPUs in large-scale computations remains as a main bottleneck. This is a major challenge supercomputing has been facing over the last decade [6]. Large parallel FFT is well-known to be communication bounded. Experiments and models have shown that for large node counts the impact of communication needs to be efficiently managed to properly target exascale systems [5, 15].

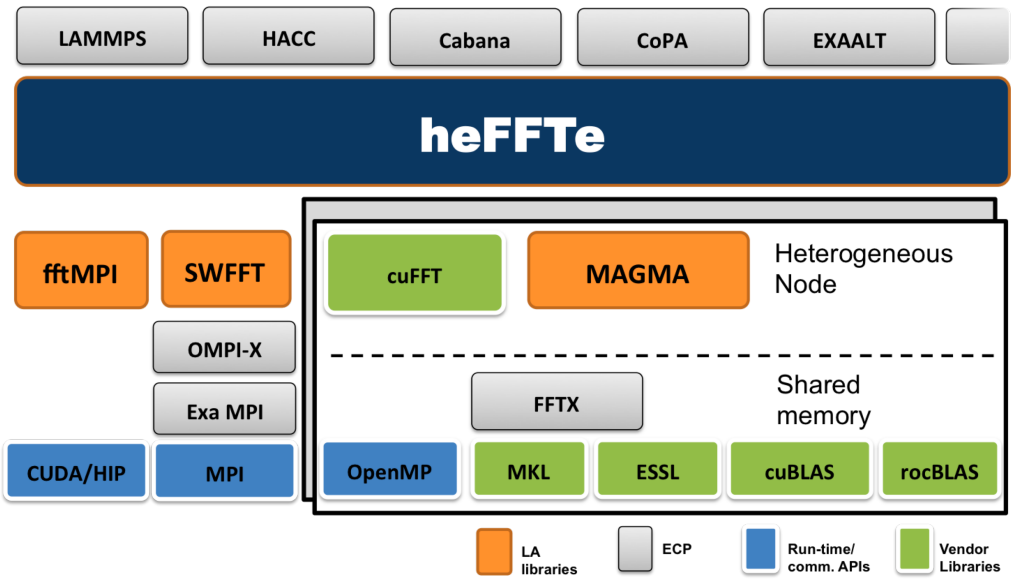


Figure 3.1: *heFFTe* in the Exascale Computing Project (ECP) software stack.

We developed and released the *heFFTe* version 0.2 library that provides very good strong and weak scalability for large node count. *heFFTe* is open-source and consists of C++ and CUDA kernels with CUDA-aware MPI communications. It is publicly available [2] and well documented [23, 26–28]. Its main objective is to become the standard for large FFT computations on the upcoming exascale systems. Figure 3.1 shows how *heFFTe* is positioned on the ECP software stack, and some of its target exascale applications (illustrated, e.g., in the gray boxes on top).

## 3.1 Methodology and Algorithmic Design

Multidimensional FFTs can be performed by a sequence of low-dimensional FFTs (see, e.g., [12]). Typical approaches used by parallel libraries are the *pencil* and *slab* decompositions. Algorithm 1 presents the pencil decomposition approach, which computes 3D FFTs by means of three 1D FFTs. This approach is schematically shown in Fig. 3.2. On the other hand, slab decomposition relies on computing sets of 2D and 1D FFTs.

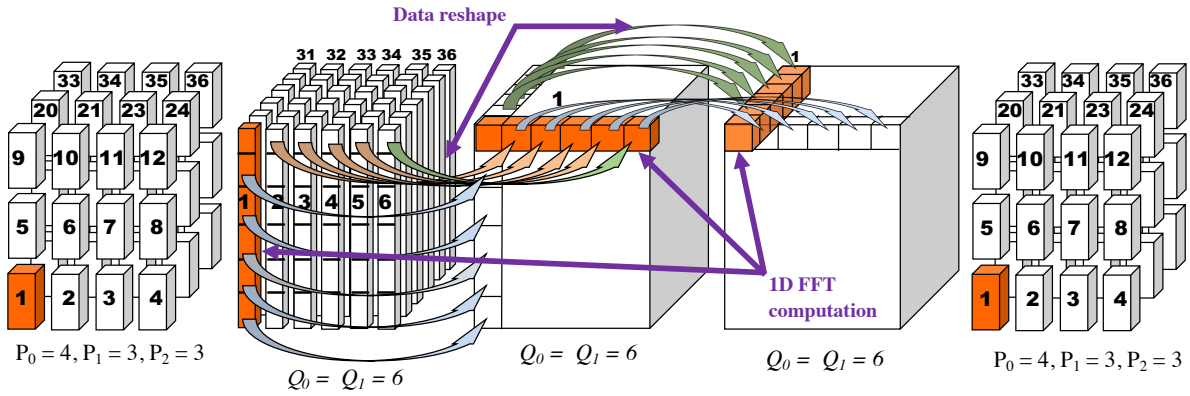


Figure 3.2: 3D FFT with pencil decomposition, schematic version of Algorithm 1.

**Algorithm 1** 3D FFT algorithm via pencil decomposition approach

**Require:** Initial and final processor grid $P_0 \times P_1 \times P_2$.
Data in spatial domain, $N_0/P_0 \times N_1/P_1 \times N_2/P_2$
**Ensure:** FFT transform in frequency domain, $\widehat{N_0}/P_0 \times \widehat{N_1}/P_1 \times \widehat{N_2}/P_2$.

Calculate a 2D grid $Q_0$ and $Q_1$ s.t. $Q_0 \times Q_1 = P_0 \times P_1 \times P_2$.

$$
\begin{array}{lcl}
N_0/P_0 \times N_1/P_1 \times N_2/P_2 & \xrightarrow{\text{Reshape}} & N_0 \times N_1/Q_0 \times N_2/Q_1 \\
N_0 \times N_1/Q_0 \times N_2/Q_1 & \xrightarrow{\text{First Dimension 1D FFTs}} & \widehat{N_0} \times N_1/Q_0 \times N_2/Q_1 \\
\widehat{N_0} \times N_1/Q_0 \times N_2/Q_1 & \xrightarrow{\text{Reshape}} & \widehat{N_0}/Q_0 \times N_1 \times N_2/Q_1 \\
\widehat{N_0}/Q_0 \times N_1 \times N_2/Q_1 & \xrightarrow{\text{Second Dimension 1D FFTs}} & \widehat{N_0}/Q_0 \times \widehat{N_1} \times N_2/Q_1 \\
\widehat{N_0}/Q_0 \times \widehat{N_1} \times N_2/Q_1 & \xrightarrow{\text{Reshape}} & \widehat{N_0}/Q_0 \times \widehat{N_1}/Q_1 \times N_2 \\
\widehat{N_0}/Q_0 \times \widehat{N_1}/Q_1 \times N_2 & \xrightarrow{\text{Third Dimension 1D FFTs}} & \widehat{N_0}/Q_0 \times \widehat{N_1}/Q_1 \times \widehat{N_2} \\
\widehat{N_0}/Q_0 \times \widehat{N_1}/Q_1 \times \widehat{N_2} & \xrightarrow{\text{Reshape}} & \widehat{N_0}/P_0 \times \widehat{N_1}/P_1 \times \widehat{N_2}/P_2
\end{array}
$$

In Algorithm 1, $\hat{N}_i$ denotes output data obtained from applying 1D FFT of size $N_i$ on the $i$-th direction. This approach can be summarized as follows, the input data of size $N_0 \times N_1 \times N_2$ is initially distributed into a grid of $P$ processors, $P = P_0 \times P_1 \times P_2$, in what is known as *brick* decomposition. Then, a reshape

(transposition) puts data into pencils on the first direction where the first set of 1D FFTs are performed. These two steps are repeated for the second and third direction. Observe that intermediate reshaped data is handled in new processor grids which must be appropriately created to ensure load-balancing. For simplicity, a single $Q_0 \times Q_1$ grid is used in Algorithm 1. Finally, a last data-reshape takes pencils on the third direction into the output brick decomposition. Note that the input and output grid of processors can be different, and that data marked by the orange boxes in Fig. 3.2 belongs to a same processor.

Several applications, e.g., in molecular dynamics, typically provide data on brick shape and require output on brick shape as well. Hence, four data reshapes, as shown in Fig. 3.2, are performed. This is what the FFTMPI [18] and *heFFTe* libraries support by default. On the other hand, several applications have their input distributed on pencils on the first direction and require the FFT output written as pencils on the third direction. This approach only requires two data-reshapes, and is the default for SWFFT, FFTE [25] and AccFFT [11] libraries. *heFFTe* also supports these options as they are used in many applications, e.g., HACC.

## 3.2 Kernels implementation

Two main sets of kernels interleave into a parallel FFT computation:

1. *Computation of low dimensional FFTs*, which can be obtained by optimized libraries for single node FFT, as those described in Chapter 2.
2. *Data reshape*, which essentially consists of a tensor transposition, and takes a great part of the computation time on CPUs, in particular.

To compute low-dimensional FFTs, *heFFTe* supports the use of several open-source as well as vendor libraries for single node, e.g., as those described in Chapter 2. *heFFTe* also provides templates for data-types and functions that help users to easily add new libraries for this purpose.

Data reshape is essentially built with two sets of routines. The first one consists of data *packing* and data *unpacking* kernels which, respectively, manage data to be sent and to be received among processors. Generally, these sets of kernels account for less than 10% of the reshaping time. Several options for packing and unpacking data are available in *heFFTe*, and there is an option to tune and find the best one for a given problem on a given architecture. The second set of routines corresponds to communication kernels. *heFFTe* supports binary and collective communications as presented in Table 3.1, with tuning tools and an improved all-to-all communication kernel, c.f. Fig. 3.3.

## 3.3 Communication design and optimization

Parallel FFT libraries typically handle communication by moving data structures in the shape of pencils, bricks, or slabs of data. For each of these options the total amount of data communicated is always the same. Hence, decreasing the number of messages between processors yields to increasing the size of the messages they send. On the other hand, for modern hardware architectures, it is well-known that latency and bandwidth improvements do not grow as quickly as the arithmetic computation power [6]. Therefore, it is important to choose the appropriate communication scheme. For instance, reshaping bricks to pencils data requires $O(P^{1/3})$ messages; this can be verified by overlapping both grids. Analogously, the number of messages for reshaping pencils in one directions to pencils in another direction is $O(P^{1/2})$, while $O(P^{2/3})$ for bricks to slabs, and $O(P)$ for slabs to pencils.

Choosing the right communication scheme highly depends on the problem size and hardware features. *heFFTe* support these options, and performing them using MPI_Alltoallv within subgroups generally yields to better performance. However, optimizations of all-to-all routines on heterogeneous clusters are still not available (e.g., in the NVIDIA Collective Communications Library [3]), even though, as can be seen from Fig. 5.3, improvements to all-to-all communication are critical. For this reason, we developed a routine called `heFFTe_Alltoallv` which includes several all-to-all communication kernels and can be used for tuning and selecting the best one for a given architecture. This routine includes a non-blocking MPI scheme mixed with CUDA IPC memory handlers which shows faster communication (up to 32 nodes) compared to classical MPI calls, as shown in Fig. 3.3.

Table 3.1: MPI routines required by parallel FFT libraries.

| Libraries | Point-to-point routines | | Collective routines | | Process Topology |
|---|---|---|---|---|---|
| | Blocking | Non-blocking | Blocking | Non-blocking | |
| FFTMPI | MPI_Send | MPI_Irecv | MPI_Allreduce<br>MPI_Allttoallv | None | MPI_Group<br>MPI_Comm_create |
| SWFFT | MPI_Sendrecv | MPI_Isend<br>MPI_Irecv | MPI_Allreduce<br>MPI_Barrier | None | MPI_Cart_create<br>MPI_Cart_sub |
| AccFFT | MPI_Sendrecv | MPI_Isend<br>MPI_Irecv | MPI_Alltoallv<br>MPI_Bcast | None | MPI_Cart_create |
| FFTE | None | None | MPI_Alltoallv<br>MPI_Bcast | None | None |
| *heFFTe* | MPI_Send<br>MPI_Recv<br>MPI_Sendrecv | MPI_Isend<br>MPI_Irecv | MPI_Allttoallv<br>MPI_Allreduce<br>MPI_Barrier | heFFTe_Alltoallv | MPI_Comm_create<br>MPI_Group<br>MPI_Cart_sub |

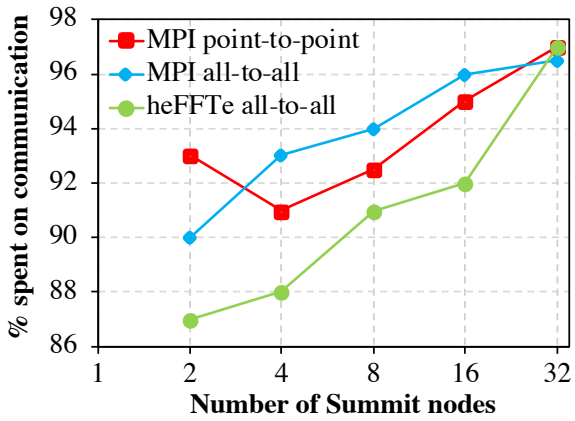


Figure 3.3: Comparing heFFTe_alltoall to MPI standard routines.

Finally, we investigated how to optimize the communication on multi-GPU and multi-lane systems like Summit [4]. We propose a modification to MPI routines for a better management of multi-rail communication. Although this approach could be very useful, we observed that the cost of such management negatively impacts the performance, degrading the potential benefit of the multi-rail optimization.

# CHAPTER 4

# Multi-node communication model

To describe the bottlenecks that FFT computations face while targeting exascale, communication models for different type of cluster architectures can be deduced and experimentally verified [5]. These models can be built for specific communication frameworks, e.g., as for pencil and slab data exchanges [24]; or they could be oriented to the hardware architecture [11].

In this section, we propose an inter-node communication model for large FFTs. We focus on inter-node effects since faster interconnection is typically available for intra-node communications, e.g., through NVLINK. Furthermore, properly scheduling intra-node communications can overlap their cost with the inter-node communications. In Table 4.1, we summarize the parameters to be used for the communication model.

To create a communication model, we analyze the *computational intensity* ($\varphi$) in Flops/Byte. For the case of FFT, we have that the number of FLOPS is $5N\log(N)$ and the volume of data moved at each reshape is $\alpha N$. Then, for the total FFT computation using $n$ nodes, we get,

$$\varphi := n\frac{C}{M} = \frac{5n\log(N)}{\alpha r}, \tag{4.1}$$

Table 4.1: Parameters for communication model

| Symbol | Description |
|---|---|
| $N$ | Size of FFT |
| $n$ | Number of Nodes |
| $r$ | Number of reshapes (tensor transpose) |
| $\alpha$ | Size of datatype (Bytes) |
| $M$ | Message size per node (Bytes) |
| $W$ | Inter-node bandwidth (GB/s) |

and the peak performance (in GFlops) is defined as,

$$\Psi := \varphi B = \frac{5n\log(N)B}{\alpha r}. \tag{4.2}$$

For the case of Summit supercomputer, we have a node interconnection of $B = 25$ GB/s, considering $r = 4$ (c.f. Fig. 3.2) and data-type as double-precision complex (i.e. $\alpha = 16$). Then,

$$\Psi_{\text{Summit}} = \frac{5n\log(N) * 25}{16 * 4} = 1.953\, n\ \log(N). \tag{4.3}$$

Fig. 5.5 shows *heFFTe*'s performance for a typical FFT of size $N = 1024^3$, and compares it to the roofline peak for increasing the number of nodes used. The results show that *heFFTe* is getting about 90% of the roofline peak performance.

We note that CPU-based FFT libraries still not get close to the performance of the roofline model presented because local CPU computations and data movements still take a large portion of the total FFT execution time. On the other hand, *heFFTe* uses GPUs, where the local computations and data movements are accelerated about $43\times$ (see Fig. 5.3), compared to CPU nodes, which renders local computations and data reshuffles insignificant to the total FFT execution time.

# CHAPTER 5

## Scalability performance results

In this section we present numerical experiments on the Summit supercomputer at ORNL, which has 4,608 nodes, each composed by 2 IBM Power9 CPUs and 6 Nvidia V100 GPUs. For our experiments, we use the pencil decomposition approach, which is commonly available in classical libraries and can be shown to be faster than the slab approach for large node count [24]. In Fig. 6.1, we first show strong scalability comparison between *heFFTe* GPU and CPU implementations, where the former being $\sim 2\times$ faster than the latter. We observe very good linear scalability in both curves. Also, since *heFFTe* CPU version was based on improved versions of kernels from FFTMPI and SWFFT libraries [26], its performance is at least as good as them. *heFFTe* CPU improved packing and unpacking kernels by blocking transpositions similar to the GPU versions, as profiling showed that FFTMPI and SWFFT counterparts can be improved. Therefore, *heFFTe* GPU is also $\sim 2\times$ faster than FFTMPI and SWFFT libraries.

Next, Fig. 5.2 shows weak scalability comparison of *heFFTe* GPU and CPU implementations for different 3D FFT sizes, showing over $2\times$ speedup and very good scaling.

In order to show the impact of local kernels acceleration, Fig. 5.3 shows a profile of a single 3D FFT using both CPU and GPU versions of *heFFTe*, where over $40\times$ speedup acceleration of local kernels, and the great impact of communication are clearly displayed.

Next, in Fig. 5.4, we compare the strong and weak scalability of the *heFFTe* and FFTE libraries. We conclude that *heFFTe* overcomes FFTE in performance (by a factor $> 2$) and has better scalability. We do not include comparison results with the AccFFT library since its GPU version did not verify correctness on several experiments performed in Summit. However, AccFFT reported a fairly constant speedup of $\sim 1.5$ compared with FFTE, while having very similar scalability [11].

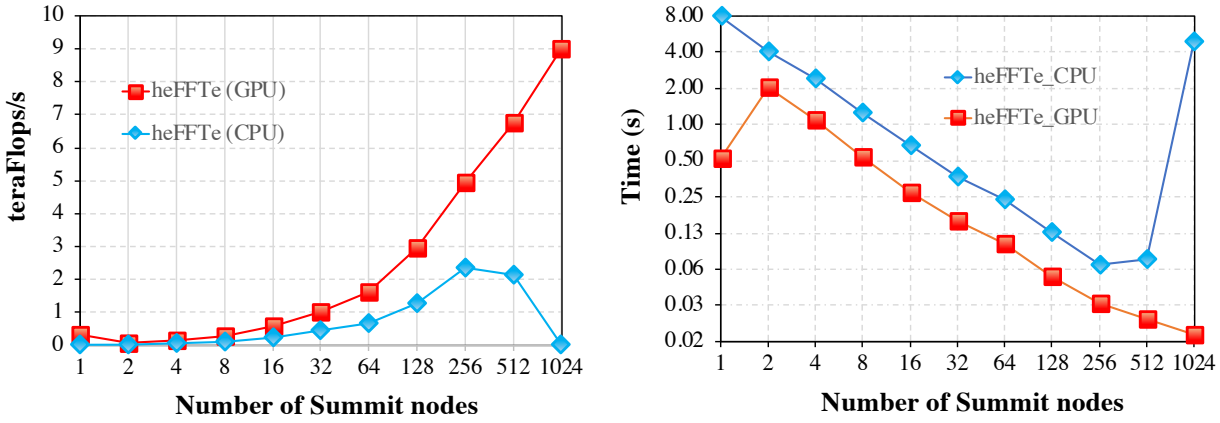


Figure 5.1: Strong scalability on 3D FFTs of size $1024^3$, using 24 MPI processes (1 MPI per Power9 core) per node (blue), and 24 MPI processes (4 MPI per GPU-V100) per node (red).

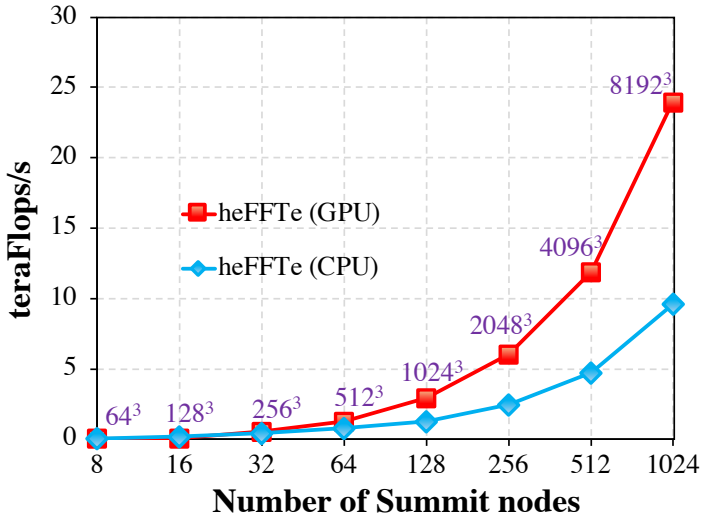


Figure 5.2: Weak scalability for 3D FFTs of increasing size, using 24 MPI processes (1 MPI per Power9 core) per node (blue), and 24 MPI processes (4 MPI per GPU-V100) per node (red).

## 5.1 Multi-node communication model

In Fig. 5.5, we numerically analyze how we approach to the roofline peak performance as described in Chapter 4. We observe that by appropriately choosing the transform size and the number of nodes, we approach to the proposed peak, and hence a correlation could be established between these two parameters to ensure that maximum resources are being used, while still leaving GPU resources to simultaneously run other computations needed by applications.

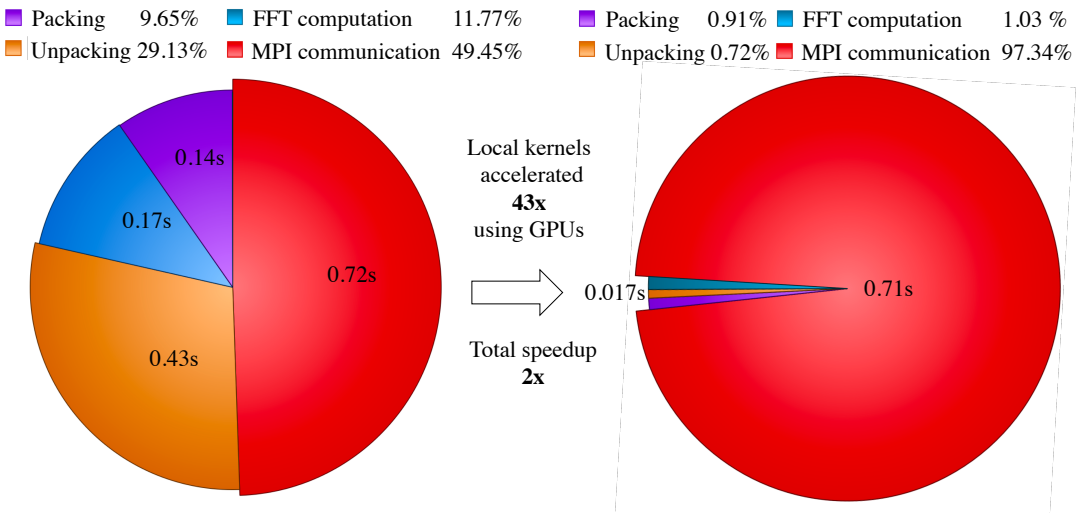


Figure 5.3: Profile of a 3D FFT of size $1024^3$ on 4 CPU nodes – using 128 MPI processes, 32 MPIs per node, 16 MPIs per socket (Left) and 4 GPU nodes – using 24 MPI processes, 6 MPIs per node, 3 MPI per socket, 1 GPU per MPI (Right)

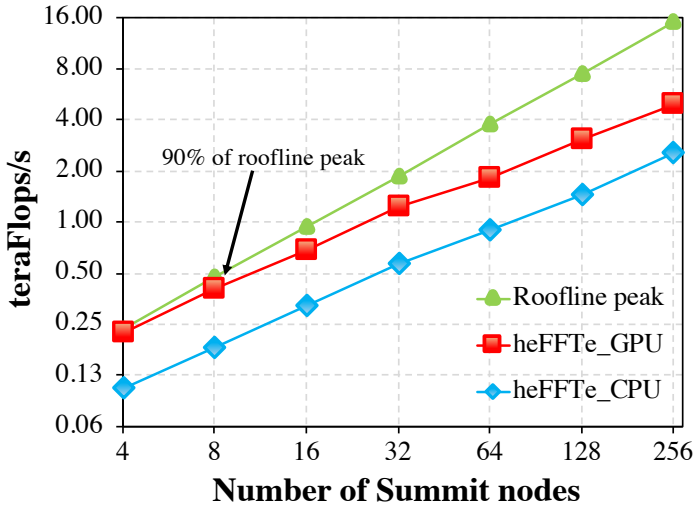


Figure 5.5: Roofline performance from Eq. 4.3 and *heFFTe* performance on a 3D FFT of size $1024^3$; using 40 MPI processes, 1MPI/core, per node (blue), and 6 MPI/node, 1MPI/1GPU-Volta100, per node (red).

## 5.2 Using *heFFTe* with applications

Diverse applications targeting exascale make use of FFT within their models. In this section, we consider LAMMPS [14], part of the EXAALT ECP project. Its KSPACE package provides a variety of long-range Coulombic solvers, as well as pair styles which compute the corresponding pairwise Coulombic interactions. This package heavily rely on efficient FFT computations, with the purpose to compute the energy of a molecular system.

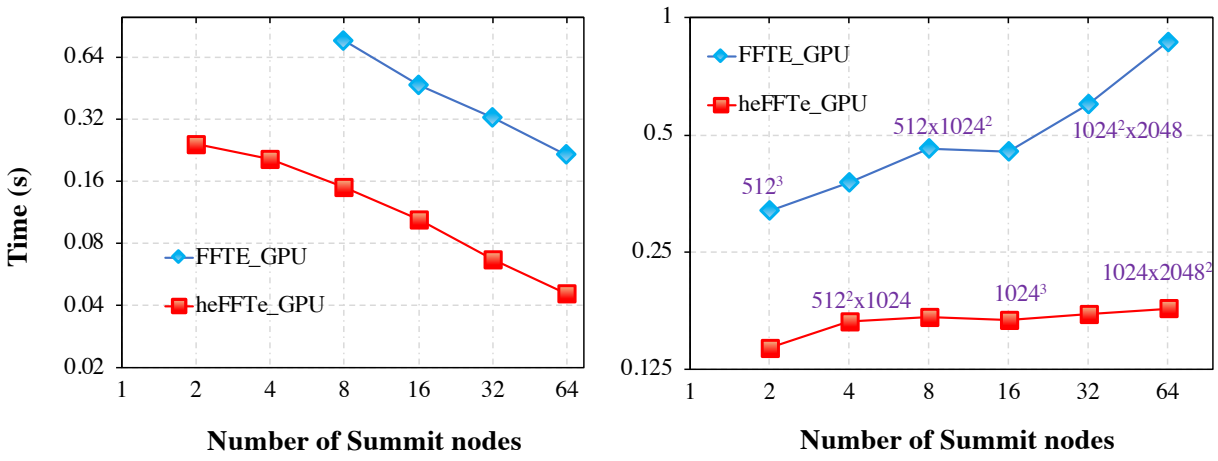


Figure 5.4: Strong scalability for a $1024^3$ FFT (left), and weak scalability comparison (right). Using 40 MPI processes, 1MPI/core, per node (blue), and 6 MPI processes with 1MPI/GPU-Volta100 per node (red).

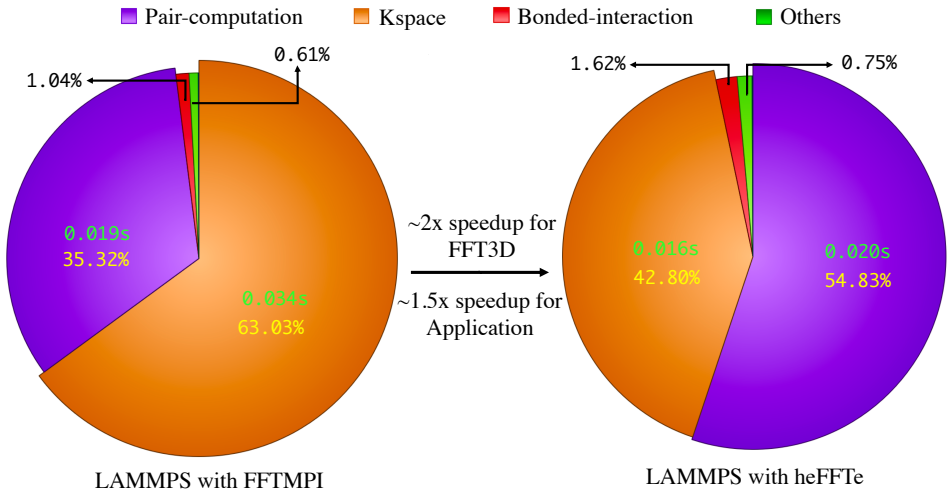


Figure 5.6: LAMMPS Rhodopsin protein benchmark on a $128^3$ FFT grid, using 2 nodes, 4 MPI processes per node. For FFTMPI we use 1 MPI per core plus 16 OpenMP threads, and for *heFFTe* we use 1 MPI per GPU.

In Fig. 5.6 we present an experiment obtained using a LAMMPS benchmark experiment, where we compare the performance when using its built-in FFTMPI library, and then using the GPU version of *heFFTe* library. As shown in Fig. 6.1, it is expected that even for large runs, using LAMMPS with *heFFTe* would provide a $2\times$ speedup of its KSPACE routine.

# CHAPTER 6

# The heFFTe version 0.2 release

We released *heFFTe* version 0.2. *heFFTe* relies on MPI for communications, OpenMP for multithreading, CUDA and HIP for GPU acceleration, and also laverages existing FFT capabilities, including options from fftMPI and SWFFT, and 1-D FFTs from vendors or open sorce efforts, etc. For NVIDIA GPUs, *heFFTe* uses the CUDA runtime, the cuFFT library, MAGMA, and hand-coded CUDA kernels.

*heFFTe* supports different precisions and uses MAGMA to accelerate various linear algebra and auxiliary kernels. This includes matrix transpositions, data reshufles for packing/unpacking of data for MPI communications, and casting routines for the development of mixed-precision algorithms. *heFFTe* supports all options from fftMPI and SWFFT, e.g., input data can be bricks or pencils, output data can be bricks or pencils. All operations in *heFFTe* are done on GPUs when using GPUs, and communications use CUDA-Aware MPI with GPU-Direct communication technologies.

New additions include:

- Algorithmic and optimizations work for reduced communication;
- Application-specific FFT optimizations;
- Multiprecision algorithms;
- R2C transforms.

*heFFTe* has very good strong scalability, which is essential in order to scale the solution time down of relatively small problems by increasing the amount of computational resources used. We ran scalability tests on up to 512 nodes (3,072 V100 GPUs) on Summit for 3-D FFT problems of size $1024^3$. Results are summarized in Figure 6.1.

Note that even with that amount of GPUs used *heFFTe* scales very well, while the corresponding CPU solution reaches the limits of its strong scalability, and even shows some slowdown. The GFlop/s reported are for double complex precision calculations, starting from bricks/cubes data distribution and ending with the original bricks/cubes data distribution. Flops are counted as $15 * 1024^3 * log_2 1024$.

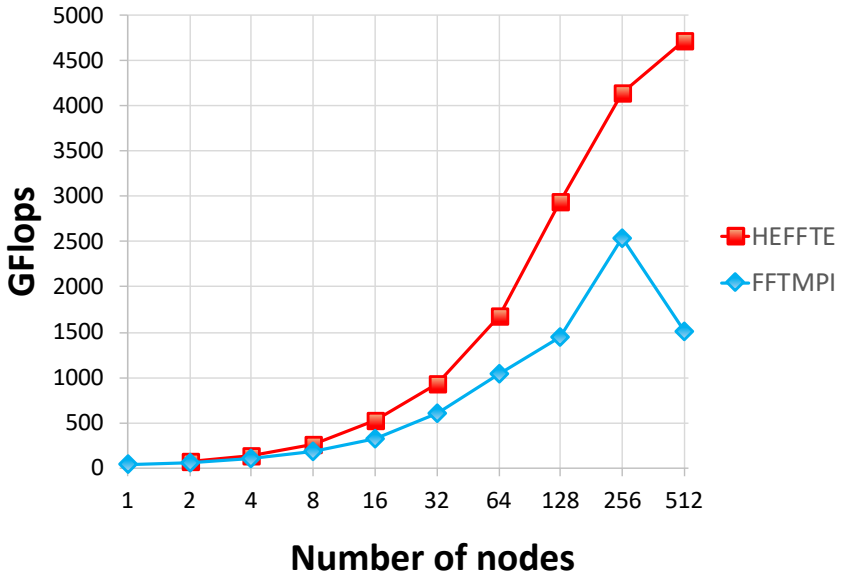


Figure 6.1: Strong scalability and performance comparison of 3-D FFTs of size $1024^3$ on up to 512 nodes of Summit supercomputer: FFTMPI using 40 cores per node and heFFTe using 6 V100 GPUs per node.

## 6.1 Application programming interface (API) for heFFTe

The software design of *heFFTe* aims to be portable among several architectures, and once it is built, applications can easily link to it and start calling the *heFFTe* kernels.

### 6.1.1 Initialization

For using *heFFTe*, one needs to run first a standard initialization kernel. This can be done while starting MPI. We support FFT computations within a sub-communicator, `fft_comm`, which must be provided by the user. The initialization API is illustrated with the following code:

```
#include <heffte.h>

int main(int argc, char *argv[]) {

    MPI_Init(&argc, &argv);
    MPI_Comm fft_comm = MPI_COMM_WORLD;

    heffte_init();

    ...

}
```

### 6.1.2 Define datatype and FFT object

Data can be of type real or complex, and *heFFTe* supports these types either in single (32 bits) or double (64 bits) precision. Defining an input/output array is essential and an FFT object is created to handle FFT operations on data:

```

    float *work;  /* Single precision input */
    FFT3d <float> *fft = new FFT3d <float> (fft_comm);
```

### 6.1.3 Creating a plan for FFT

One of its main kernels allows *heFFTe* to create FFT plans, which determine the sequence of steps to follow to go from input through output.

A plan can be created for a processor grid where each of the processors has been assigned a brick of the input data, c.f., 3.2. If the user only provides input data and number of computational resources, then *heFFTe* provides kernels to create the processors and the data grids, as follows:

```

    /* Create grid of processors */
    heffte_proc_setup(N, proc_i, nprocs);
    heffte_proc_setup(N, proc_o, nprocs);

    /* Create grid of data (local bricks) */
```

```
heffte_grid_setup(N, i_lo, i_hi, o_lo, o_hi,
                  proc_i, proc_o, me, nfft_in, nfft_out);

/* Create FFT plan */
heffte_plan_create(dim, work, fft, N, i_lo, i_hi, o_lo, o_hi, permute, workspace);
```

To create the processor grid, the user only needs to provide the number of MPI processors to use (`nprocs`). To create the data grid, the user must define the 6 vertices of the local brick data at input (`i_lo, i_hi`) and output (`o_lo, o_hi`), as showed on Fig. 6.2. The user then receives the local FFT size at input (`nfft_in`) and output (`nfft_out`).

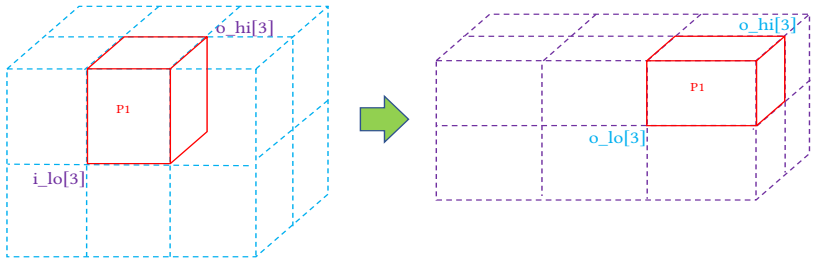


Figure 6.2: Local brick at input (left) and output (right) for processor P1.

Then, once grids are defined, user can create an FFT plan by simply providing:

- `dim`: Problem dimension, e.g., `dim=3`
- `N` : Array size, e.g., `N=[nx,ny,nz]`
- `permute` : Permutation storage of output array

After creating a plan, user is ready to execute FFTs, and memory requirements are returned in `workspace` array.

### 6.1.4 Setting memory type and allocation

Memory allocation must carefully be chosen, according to where the data must be processed (CPUs/GPUs). *heFFTe* provides 7 types of memory that user can choose. We provide efficient kernels for safe allocation. If input array has not been defined, then it must be allocated and initialized.

```
/* Allocate memory */
heffte_allocate(HEFFTE_MEM_CPU, &work, workspace[0], nbytes);
fft-> mem_type = HEFFTE_MEM_CPU;

/* Initialize vector, with random numbers using a seed value */
heffte_initialize_host(work, nfft_in, seed, HEFFTE_COMPLEX_DATA);
```

### 6.1.5 Execution

One of most important kernels available in *heFFTe* is the one in charge of the FFT computation. This kernel has the same syntax for any type of data and its usage follows APIs from CUFFT and FFTW3.

```
30  /* Compute an inplace C2C forward FFT */
31  heffte_execute(fft, work, work, FORWARD);
```

Similar execution function is available for the case of the R2C transforms, `heffte_execute_r2c`, which is available with the *heFFTe* 0.2 release.

## 6.2 heFFTe profiler

For users that require a detailed trace of the FFT computation, *heFFTe* provides a default profiler kernel that can trace the *heFFTe* kernels as presented below.

```
32  heffte_tracing("start");
33
34  /* Compute an inplace C2C forward and backward FFT */
35  heffte_execute(fft, work, work, BACKWARD);
36  heffte_execute(fft, work, work, BACKWARD);
37
38  heffte_tracing("stop"));
39
40  /* Finalize code */
41  delete fft;
42  heffte_deallocate(HEFFTE_MEM_CPU, work);
43  MPI_Finalize();
```

When `heffte_tracing` is called, it will produce a `svg` file containing a detailed timing for the section of code, e.g., see Fig. 6.3.

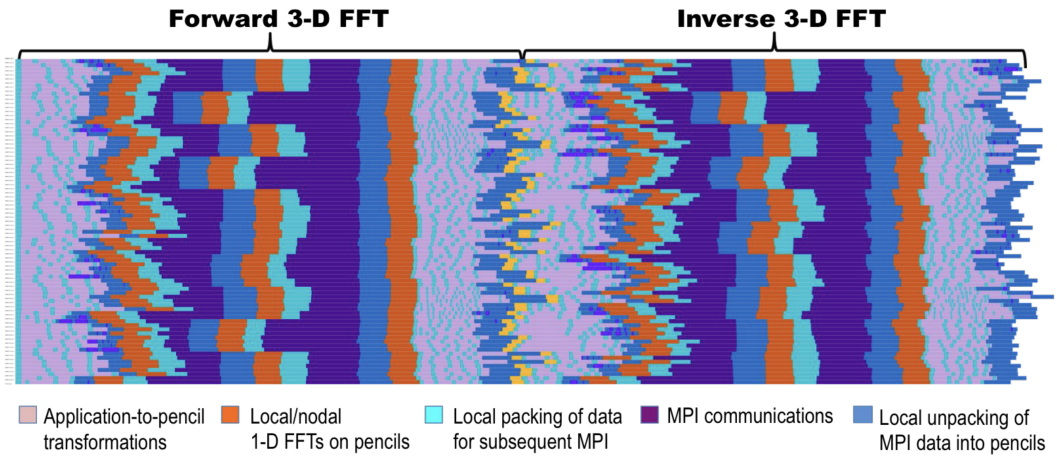


Figure 6.3: Tracing with *heFFTe*'s default profiler.

## 6.3 Tracing heFFTe with vendor libraries

Detailed profiling of *heFFTe* functions can be obtained also using vendor libraries such as the NVIDIA Visual Profiler, TAU, ScoreP and Vampir.

The version 0.2 of *heFFTe* includes a Vampir filter, `heFFTe_filter.filt`, which together with a profiler script, `vampir_heFFTe.sh` can trace kernels by simply adding a prefix to *heFFTe* execution, e.g.,

```
mpirun -np 2 ./vampir_trace.sh ./heffte_exec -my_options ...
```

Once this profiler is called, it will produce `.otf2` files, which provide detailed information and several graphs using Vampir interface, e.g., see Fig. 6.4.

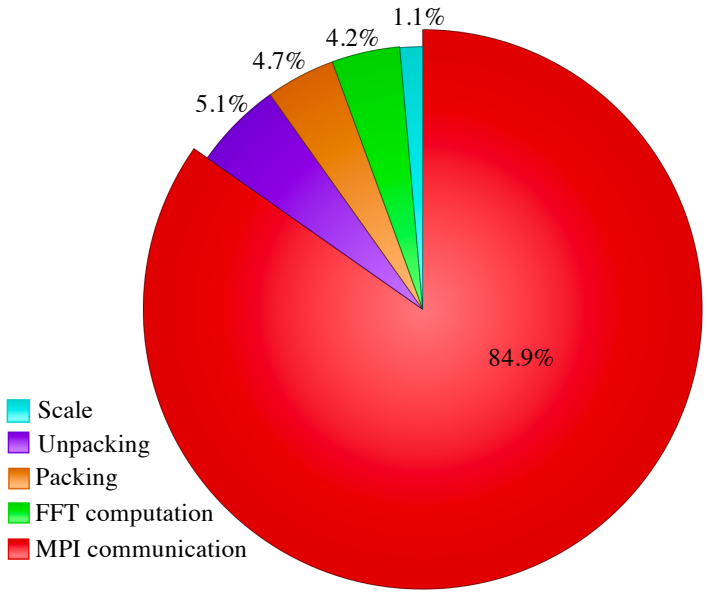


Figure 6.4: Tracing *heFFTe* using Vampir.

.

# CHAPTER 7

# Conclusions and future work directions

In this milestone, we implementated a high-performance prototype for multidimensional FFTs, optimized performance on GPUs, and presented performance and scalability results. *heFFTe* version 0.2 was released. We provided experiments showing considerable speedups compared to state-of-the-art libraries, and that linear scalability is achievable. We have greatly accelerated local kernels using GPUs and are currently getting very close to the experimental roofline peak on the Summit supercomputer. Our results show that further optimizations would require better hardware interconnection and/or new communication-avoiding algorithmic approaches.

Future work will concentrate on MPI optimizations for strong scaling on many nodes, optimizations for a single node for cross-socket communications, and algorithmic optimizations based on slab partitions, or other reductions of the computational resources used that can lead to reduced communications. More versions and support for different FFT features are being added in FFT-ECP. Currently, local nodal computations and data reshuffles (packing and unpacking) are accelerated about $43\times$ using GPUs, compared to CPUs. This makes the MPI communications the main bottleneck, as now $96\%$ of the time is in MPI. Thus, the GPU is not used $96\%$ of the time, opening possibilities for application developers to know that this is available to them and develop new algorithms that use the GPUs. Therefore, future efforts will also be on application-specific optimizations. Furthermore, the use of mixed-precision calculations [**?** ] is also becoming of increased interest, especially for GPUs, where for example the GPUs can be used to compress data for additional acceleration coming from the reduced communications (with compressed data). We have started investigating the use of various data compressions for reduced communications (including lossy compression).

# Acknowledgments

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations (the Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

# Bibliography


[1] AMD library for FFT computation. URL https://github.com/clMathLibraries/clFFT.

[2] HEFFTE library. URL https://bitbucket.org/icl/heffte.

[3] NVIDIA library for collective communication. URL https://github.com/NVIDIA/nccl.

[4] Alan Ayala, Xi Luo, Stanimire Tomov, Hejer Shaiek, Azzam Haidar, George Bosilca, and Jack Dongarra. Impacts of Multi-GPU MPI Collective Communications on Large FFT Computation. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*, 2019.

[5] Kenneth Czechowski, Chris McClanahan, Casey Battaglino, Kartik Iyer, P.-K Yeung, and Richard Vuduc. On the communication complexity of 3D FFTs and its implications for exascale. 06 2012. doi: 10.1145/2304576.2304604.

[6] J. Demmel. Communication-avoiding algorithms for linear algebra and beyond. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013.

[7] JD Emberson, N. Frontiere, S. Habib, K. Heitmann, A. Pope, and E. Rangel. Arrival of First Summit Nodes: HACC Testing on Phase I System. Technical Report MS ECP-ADSE01-40/ExaSky, Exascale Computing Project (ECP), 2018.

[8] Salvatore Filippone. The IBM parallel engineering and scientific subroutine library. In Jack Dongarra, Kaj Madsen, and Jerzy Waśniewski, editors, *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 199–206, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-49670-0.

[9] Franz Franchetti, Daniele Spampinato, Anuva Kulkarni, Doru Thom Popovici, Tze Meng Low, Michael Franusich, Andrew Canning, Peter McCorquodale, Brian Van Straalen, and Phillip Colella. FFTX and SpectralPack: A First Look. *IEEE International Conference on High Performance Computing, Data, and Analytics*, 2018.

[10] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[11] Amir Gholami, Judith Hill, Dhairya Malhotra, and George Biros. AccFFT: A library for distributed-memory FFT on CPU and GPU architectures. *CoRR*, abs/1506.07933, 2015. URL http://arxiv.org/abs/1506.07933.

[12] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Accuracy and stability of numerical algorithms*. Addison Wesley, second ed., 2003.

[13] Intel. Intel Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/. URL https://software.intel.com/en-us/mkl/features/fft.

[14] Large-scale Atomic/Molecular Massively Parallel Simulator. Large-scale atomic/molecular massively parallel simulator, 2018. Available at https://lammps.sandia.gov/.

[15] Myoungkyu Lee, Nicholas Malaya, and Robert D. Moser. Petascale direct numerical simulation of turbulent channel flow on up to 786k cores. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[16] S. Lin, N. Liu, M. Nazemi, H. Li, C. Ding, Y. Wang, and M. Pedram. Fft-based deep learning deployment in embedded systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1045–1050, 2018.

[17] CUDA NVIDIA. CUFFT library, 2018. URL http://docs.nvidia.com/cuda/cufft.

[18] parallel 2d and 3d complex FFTs. parallel 2d and 3d complex ffts, 2018. Available at http://www.cs.sandia.gov/~sjplimp/download.html.

[19] Steven Plimpton, Axel Kohlmeyer, Paul Coffman, and Phil Blood. fftMPI, a library for performing 2d and 3d FFTs in parallel. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2018.

[20] Steven J. Plimpton. FFTs for (mostly) particle codes within the doe exascale computing project, 2017.

[21] Thom Popovici, Tze-Meng Low, and Franz Franchetti. Large bandwidth-efficient FFTs on multicore and multi-socket systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018.

[22] DF Richards, O Aziz, Jeanine Cook, Hal Finkel, et al. Quantitative performance assessment of proxy apps and parents. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2018.

[23] Hejer Shaiek, Stanimire Tomov, Alan Ayala, Azzam Haidar, and Jack Dongarra. GPUDirect MPI Communications and Optimizations to Accelerate FFTs on Exascale Systems. Extended Abstract icl-ut-19-06, 2019-09 2019.

[24] D. Takahashi. Implementation of Parallel 3-D Real FFT with 2-D decomposition on Intel Xeon Phi Clusters,. In *13th International conference on parallel processing and applied mathematics.*, 2019.

[25] Daisuke Takahashi. FFTE: A fast Fourier transform package. *http://www.ffte.jp/*, 2005.

[26] Stanimire Tomov, Azzam Haidar, Daniel Schultz, and Jack Dongarra. Evaluation and Design of FFT for Distributed Accelerated Systems. ECP WBS 2.3.3.09 Milestone Report FFT-ECP ST-MS-10-1216, Innovative Computing Laboratory, University of Tennessee, October 2018. revision 10-2018.

[27] Stanimire Tomov, Azzam Haidar, Alan Ayala, Daniel Schultz, and Jack Dongarra. Design and Implementation for FFT-ECP on Distributed Accelerated Systems. ECP WBS 2.3.3.09 Milestone Report FFT-ECP ST-MS-10-1410, Innovative Computing Laboratory, University of Tennessee, April 2019. revision 04-2019.

[28] Stanimire Tomov, Azzam Haidar, Alan Ayala, Hejer Shaiek, and Jack Dongarra. FFT-ECP Implementation Optimizations and Features Phase. Technical Report ICL-UT-19-12, 2019-10 2019.