heFFTe: Highly Efficient FFT for Exascale

Alan Ayala[1], Stanimire Tomov[1], Azzam Haidar[2], and Jack Dongarra[1,3]

[1] Innovative Computing Laboratory,
The University of Tennessee, Knoxville TN, USA
{aayala,tomov}@icl.utk.edu
[2] Nvidia Corporation, California USA
azzamhaidar@nvidia.com
[3] Oak Ridge National Laboratory, USA
University of Manchester, UK
dongarra@icl.utk.edu

Abstract. Exascale computing aspires to meet the increasing demands from large scientific applications. Software targeting exascale is typically designed for heterogeneous architectures; henceforth, it is not only important to develop well-designed software, but also make it aware of the hardware architecture and efficiently exploit its power. Currently, several and diverse applications, such as those part of the Exascale Computing Project (ECP) in the United States, rely on efficient computation of the Fast Fourier Transform (FFT). In this context, we present the design and implementation of *heFFTe* (Highly Efficient FFT for Exascale) library, which targets the upcoming exascale supercomputers. We provide highly (linearly) scalable GPU kernels that achieve more than 40× speedup with respect to local kernels from CPU state-of-the-art libraries, and over 2× speedup for the whole FFT computation. A communication model for parallel FFTs is also provided to analyze the bottleneck for large-scale problems. We show experiments obtained on Summit supercomputer at Oak Ridge National Laboratory, using up to 24,576 IBM Power9 cores and 6,144 NVIDIA V-100 GPUs.

Keywords: Exascale · FFT · Scalable algorithm · GPUs.

1 Introduction

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 [21] 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 an heterogeneous, fast and scalable parallel FFT library, with an implementation that can take advantage of novel hardware components, and efficiently exploit their benefits.

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],

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_19

which has been tuned to optimally perform in several architectures. Vendor libraries for this purpose have also been highly optimized, such is the case of MKL (Intel) [13], ESSL (IBM) [8], clFFT (AMD) [1] and CUFFT (NVIDIA) [18]. Novel libraries are also being developed to further optimize single node FFT computation, e.g. FFTX [9] and Spiral [22]. Most of the previous libraries have been extended to distributed memory versions, some by the original developers, and others by different authors.

1.1 Related work

In the realm of *distributed-CPU* libraries, FFTW supports MPI via slab decomposition, however it has limited scalability and hence it is limited to a small number of nodes. P3DFFT [19] extends FFTW functionalities and supports both pencil and slab decompositions. Large scale applications have built their own FFT library, such as FFTMPI [20] (built-in on LAMMPS [14]) and SWFFT [23] (built-in on HACC [7]). These libraries are currently being used by several molecular-dynamics applications.

Concerning *distributed-GPU* libraries, the slab-approach introduced in [17] is one of the first heterogeneous codes for large FFT computation on GPUs. Its optimization approach is limited to small number of nodes and focus on reducing tensor transposition cost (known to be the bottleneck) by exploiting infiniband-interconnection using the IBverbs library, which makes it not portable. Further improvements to scalablity have been presented in FFTE library [26] which supports pencil decompositions and includes several optimizations, although with limited features and limited improvements on communication. Also, FFTE relies on the commercial PGI compiler, which may limit its usage. Finally, one of the most recent libraries is AccFFT [11], its approach consists in overlapping computation and blocking collective communication by reducing the PCIe overhead, they provide good (sublinear) scalability results for large real-to-complex transforms using NVIDIA K20 GPUs.

Even though the fast development of GPUs has enabled great speedup on local computations, the cost of communication between CPUs/GPUs on large-scale computations remains as the bottleneck, and 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 on communication needs to be efficiently managed to properly target exascale systems [5, 15].

In this context, we introduce *heFFTe* (pronounced "*hefty*"), which provides very good (linear) scalability for large node count, it is open-source and consists of C++ and CUDA kernels with (CUDA-aware) MPI and OpenMP interface for communication. It has a user-friendly interface and does not require any commercial compiler. Wrappers to interface with C, Fortran and Python are available. It is publicly available in [2] and documented in [24, 27–29]. Its main objective is

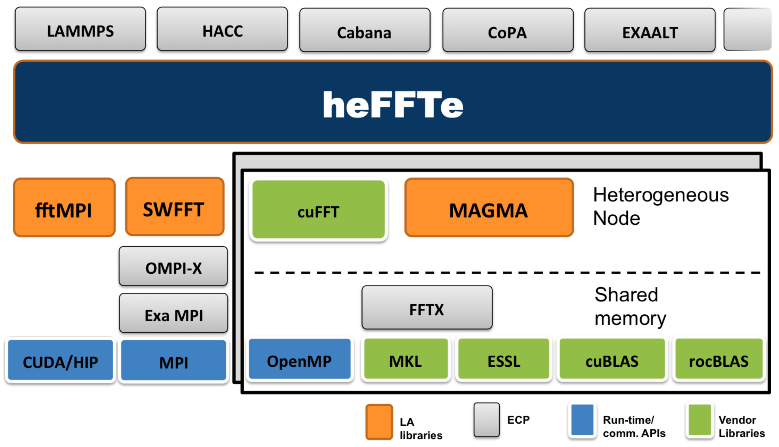

Fig. 1. *heFFTe* in the Exascale Computing Project (ECP) software stack.

to become the standard for large FFT computations on the upcoming exascale systems. Fig. 1 shows how *heFFTe* is positioned on the ECP software stack, and some of its target exascale applications (gray boxes).

This paper is organized as follows, Section 2 describes the classical FFT multi-dimensional algorithm and its implementation phases within *heFFTe*. We then present *heFFTe*'s main features and functionalities. Next, Section 3 presents a multi-node communication model for parallel FFTs and an approach to reduce its computational complexity for a small number of nodes. Section 4 presents numerical experiments on Summit supercomputer, and evaluates the multi-GPU communication impact on performance. Finally, Section 5 concludes our paper.

2 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. 2. On the other hand, slab decomposition relies on computing sets of 2D and 1D FFTs.

Fig. 2 schematically shows the steps during a 3D FFT computation in parallel, using a 3D partition of processors. On the top part of this figure, we present the pencil methodology, as described in Algorithm 1, in which $\hat{N}_i$ 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 processors, $P_{i0} \times P_{i1} \times P_{i2}$, in what is known

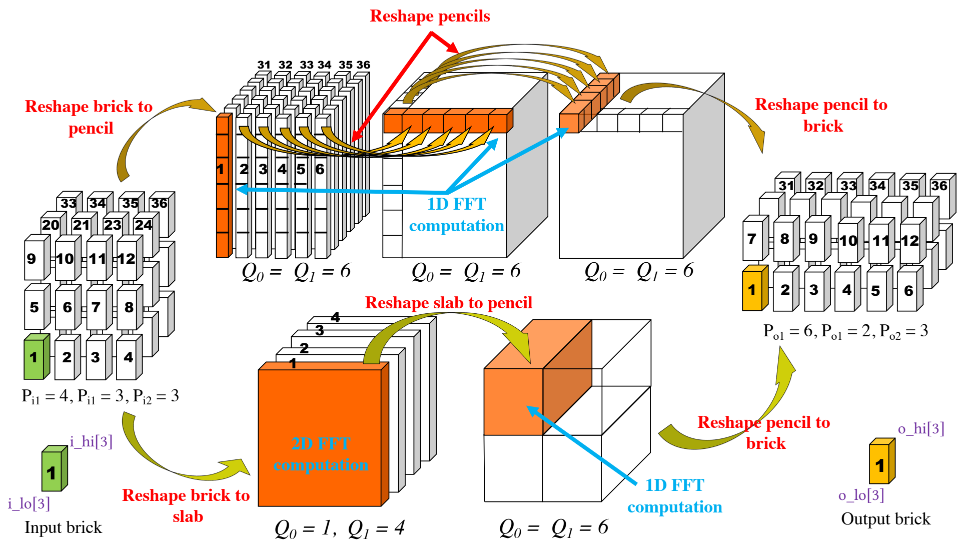

Fig. 2. 3D FFT computation steps via pencil decomposition approach (top) and via slab-pencil decomposition (bottom), c.f., Algorithm 1.

Algorithm 1 3D FFT algorithm via pencil decomposition approach

Require: Initial and final processor grids, $P_{\text{i}0} \times P_{\text{i}1} \times P_{\text{i}2} - P_{\text{o}0} \times P_{\text{o}1} \times P_{\text{o}2}$.
Data in spatial domain, $N_0/P_{\text{i}0} \times N_1/P_{\text{i}1} \times N_2/P_{\text{i}2}$

Ensure: FFT transform in frequency domain, $\widehat{N_0}/P_{\text{i}0} \times \widehat{N_1}/P_{\text{i}1} \times \widehat{N_2}/P_{\text{i}2}$.

Calculate a 2D grid Q_0 and Q_1 s.t. $Q_0 \times Q_1 = P_{\text{i}0} \times P_{\text{i}1} \times P_{\text{i}2}$.

$$\begin{array}{rcl}
N_0/P_{\text{i}0} \times N_1/P_{\text{i}1} \times N_2/P_{\text{i}2} & \xrightarrow{\text{Reshape}} & N_0 \times N_1/Q_0 \times N_2/Q_1 \\
N_0 \times N_1/Q_0 \times N_2/Q_1 & \xrightarrow{\text{First Dimension 1D FFTs}} & \widehat{N_0} \times N_1/Q_0 \times N_2/Q_1 \\
\widehat{N_0} \times N_1/Q_0 \times N_2/Q_1 & \xrightarrow{\text{Reshape}} & \widehat{N_0}/Q_0 \times N_1 \times N_2/Q_1 \\
\widehat{N_0}/Q_0 \times N_1 \times N_2/Q_1 & \xrightarrow{\text{Second Dimension 1D FFTs}} & \widehat{N_0}/Q_0 \times \widehat{N_1} \times N_2/Q_1 \\
\widehat{N_0}/Q_0 \times \widehat{N_1} \times N_2/Q_1 & \xrightarrow{\text{Reshape}} & \widehat{N_0}/Q_0 \times \widehat{N_1}/Q_1 \times N_2 \\
\widehat{N_0}/Q_0 \times \widehat{N_1}/Q_1 \times N_2 & \xrightarrow{\text{Third Dimension 1D FFTs}} & \widehat{N_0}/Q_0 \times \widehat{N_1}/Q_1 \times \widehat{N_2} \\
\widehat{N_0}/Q_0 \times \widehat{N_1}/Q_1 \times \widehat{N_2} & \xrightarrow{\text{Reshape}} & \widehat{N_0}/P_{\text{o}0} \times \widehat{N_1}/P_{\text{o}1} \times \widehat{N_2}/P_{\text{o}2}
\end{array}$$

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.

Table 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 MPI_Alltoallv	None	MPI_Group MPI_Comm_create
SWFFT	MPI_Sendrecv	MPI_Isend MPI_Irecv	MPI_Allreduce MPI_Barrier	None	MPI_Cart_create MPI_Cart_sub
AccFFT	MPI_Sendrecv	MPI_Isend MPI_Irecv	MPI_Alltoallv MPI_Bcast	None	MPI_Cart_create
FFTE	None	None	MPI_Alltoallv MPI_Bcast	None	None
heFFTe	MPI_Send MPI_Recv MPI_Sendrecv	MPI_Isend MPI_Irecv	MPI_Alltoallv MPI_Allreduce MPI_Barrier	heFFTe_Alltoall	MPI_Comm_create MPI_Group MPI_Cart_sub

Several applications provide input data already on pencil distribution on the first direction and require the output written as pencils on the third direction. In this case, only two data-reshapes are required, this is the default for FFTE [26] and AccFFT [11] libraries. On the other hand, *heFFTe* can treat input and output shapes with high flexibility, generalizing features of modern libraries, and with a friendly interface as presented in Section 2.3.

Finally, in the bottom part of Fig. 2, we show the slab approach which saves one step of data reshape by performing 2D FFTs, this has a considerable impact in performance for a small number of nodes [25].

2.1 Kernels implementation

Two main sets of kernels intervene 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 Section 1.
2. *Data reshape*, which essentially consists on a tensor transposition, and takes a great part of the computation time.

To compute low-dimensional FFTs, *heFFTe* supports several open-source and vendor libraries for single node, as those described in Section 1. And it also provides templates to select types and precision of data. For direct integration to applications, *heFFTe* provides example wrappers and templates to help users to easily link with their libraries.

Data reshape is essentially built with two sets of routines, the first one consists in *packing* and *unpacking* kernels which, respectively, manage data to be sent and to be received among processors. Generally, these set 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

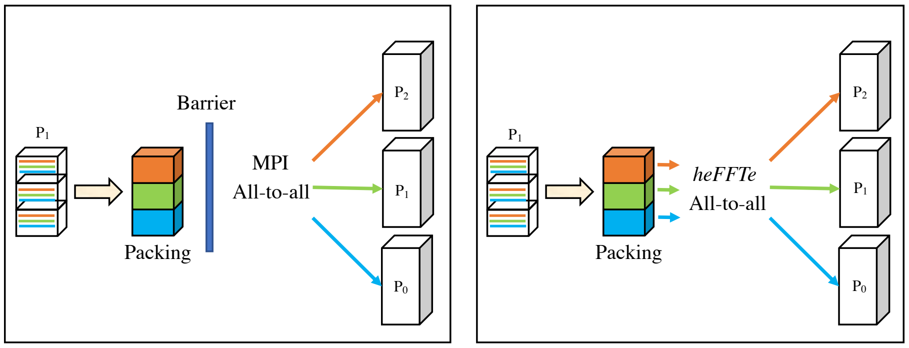

Fig. 3. Standard approach of packing and synchronous data transfer (left), and its asynchronous approach coupling packing and communication (right).

best one for a given problem on a given architecture. The last set of routines correspond to communication kernels, *heFFTe* supports binary and collective communications as presented in Table 1.

The main impact on performance obtained by *heFFTe* in comparison to standard libraries, comes from the kernels optimization ($> 40\times$ speedup w.r.t CPU libraries, c.f. Fig. 6), and also by a novel efficient asynchronously-management of packing and communication, as shown in Fig. 3, where we can observe the classical packing process supported by most libraries and a novel approach via routine *heFFTe_alltoallv*, introduced in [4], which overlaps packing/unpacking with MPI communication.

2.2 Communication design and optimization

Parallel FFT libraries typically handle communication by moving data structures on the shape of pencils, bricks, and 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 brick to pencil data requires $O(P^{1/3})$ messages, this can be verified by overlapping both grids. Analogously, the number of messages for reshaping pencil to pencil is $O(P^{1/2})$, while $O(P^{2/3})$ for brick to slab, and $O(P)$ for slab to pencil.

Choosing the right communication scheme highly depends on the problem size and hardware features, *heFFTe* supports standard MPI_Alltoallv within subgroups, which generally yields better performance compared to poin-to-point communication. However, optimizations of all-to-all routines on heterogeneous clusters are still not available (e.g. on the NVIDIA Collective Communications Library [3]), even though, as can be regarded in Fig. 6, improvements to all-to-all communication are critical. For this reason, we developed a routine called *heFFTe_alltoallv* [4] which includes several all-to-all communication kernels and can be used for tuning and selecting the best one for a given architecture. This routines aimed to provide a better management of multi-rail communication. The asynchronous approach of *heFFTe_alltoallv* was proved efficient for up to 32 nodes, and the multi-rail management, although promising, negatively impacts the performance when increasing node count, degrading the potential benefit of the multi-rail optimization [4]. Work on communication avoiding frameworks is ongoing, targeting large node count on heterogeneous clusters.

2.3 Application programming interface (API) for *heFFTe*

heFFTe's software design is built on C and C++, and provides wrappers to be used in Fortran and Python. The API aims to be user-friendly and portable among several architectures, allowing users to easily link it to their applications. The API follows styles from FFTW3 and CUFFT libraries, adding novel features, such as templates for multitype and multiprecision data.

Define FFT parameters. Distributed FFT requires data split on a processors grid. In 3D, input/output arrays are typically defined by the six vertices of a *brick*, e.g. $(i_{\mathrm{lo}}, i_{\mathrm{hi}})$, as shown in Fig. 2; *heFFTe* allows this definition using any MPI sub-communicator, `fft_comm`, which has to be provided by user together with data definition.

```cpp
#include <heffte.h>

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

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

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

FFT Plan definition. Once data and processors grids are locally defined, user can create an FFT plan by simply providing the following parameters,

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

Next, we show how a *heFFTe* plan is created; memory requirements are returned to the user as a `workspace` array.

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

Note that a single plan can be used for several FFT computations, which is typical for several applications where grids are fixed.

FFT execution. One of *heFFTe*'s most important kernels 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.

```
...
/* Compute an in-place complex-to-complex (C2C) forward FFT */
heffte_execute(fft, work, work);
```

Similar execution function is available for the case of real-to-complex (R2C) transforms, `heffte_execute_r2c`.

3 Multi-node communication model

To describe the bottleneck FFT computation 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, as for pencil and slab data exchanges [25]; 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 fast interconnection is typically available intra-node, e.g. NVLINK. And properly scheduling intra-node communications can overlap their cost with the inter-node communications. In Table 2, we summarize the parameters to be used for the communication model.

To create a communication model, we analyze the *computational intensity* (φ) 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 αN, then for the total FFT computation using P nodes, we get,

$$\varphi := P\frac{C}{M} = \frac{5P\log(N)}{\alpha r}, \tag{1}$$

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

$$\Psi := \varphi B = \frac{5P\log(N)B}{\alpha r}. \tag{2}$$

Table 2. Parameters for communication model

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

For the case of Summit supercomputer, we have a node interconnection of $B = 25$ GB/s, considering $r = 4$ (c.f. Fig. 2) and data-type as double-precision complex (i.e. $\alpha = 16$). Then,

$$\Psi_{\text{Summit}} = \frac{5P\log(N) * 25}{16 * 4} = 1.953P\log(N). \tag{3}$$

Fig. 8, shows *heFFTe*'s performance for a typical FFT of size $N = 1024^3$, and compares it to the roofline peak for increasing number of nodes, getting about to 90% close to peak value.

4 Numerical Experiments

In this section we present numerical experiments on Summit supercomputer, 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 [25]. In Fig. 4, we first show strong scalability comparison between *heFFTe* GPU and CPU implementations, being the former $\sim 2\times$ 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 [29], then its performance is at least as good as them. Therefore, *heFFTe* GPU is also $\sim 2\times$ faster than FFTMPI and SWFFT libraries. Drop in performance for the CPU implementation after 512 nodes (12,288 cores) is due to latency impact, which we verified with several experiments. This is because for a 1024^3 size, the number of messages become very large while their size becomes very small. When increasing the problem size, the CPU version keeps scaling very well as shown in Fig. 5.

Next, Fig. 5 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. 6 shows a profile of a single 3D FFT using both, CPU and GPU, versions of *heFFTe*, where over $40\times$ speedup of local kernels and the great impact of communication are

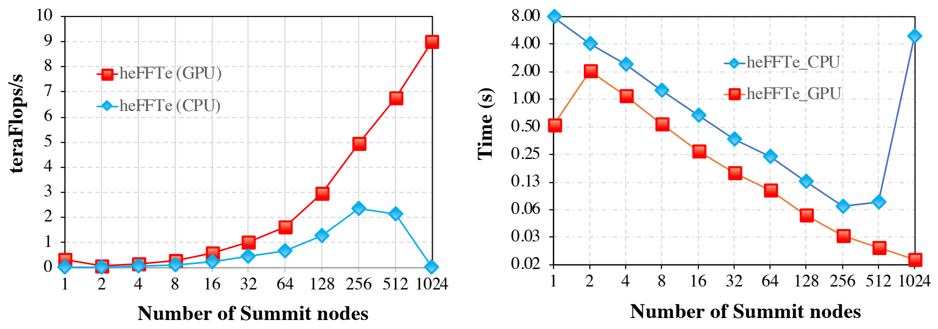

Fig. 4. Strong scalability on 3D FFTs of size 1024^3, using 24 MPI processes (1 MPI per Power9 core) per node (blue), and 24 MPI processes (4 MPI per GPU-V100) per node (red).

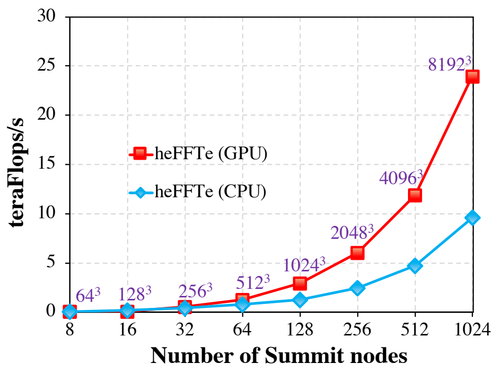

Fig. 5. Weak scalability for 3D FFTs of increasing size, using 24 MPI processes (1 MPI per Power9 core) per node (blue), and 24 MPI processes (4 MPI per GPU-V100) per node (red).

clearly displayed.

Next, in Fig. 7, we compare strong and weak scalability of *heFFTe* and FFTE libraries. Concluding that *heFFTe* overcomes FFTE in performance (by a factor > 2) and having better scalability. We do not include results with AccFFT library, since its GPU version did not verify correctness on several experiments performed in Summit. However, AccFFT reported a fairly constant speedup of ~ 1.5 compared with FFTE, while having very similar scalability [11].

4.1 Multi-node communication model

In Fig. 8, we numerically analyze how we approach to the roofline peak performance as described in Section 3. We observe that by appropriately choosing the

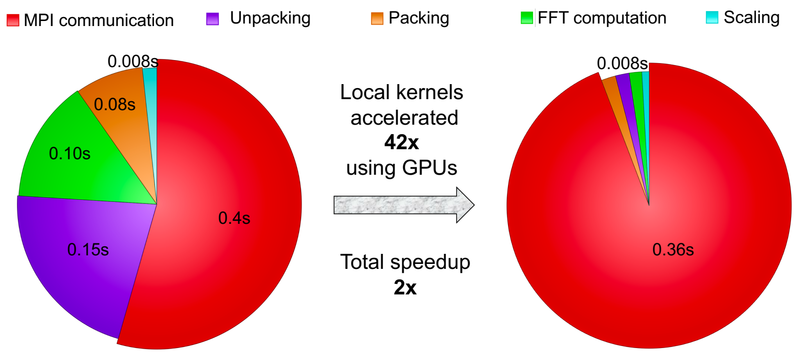

Fig. 6. Profile of a 3D FFT of size 1024^3 on 32 Summit nodes with all-to-all communication – using 40 MPI processes per node (left) and 6 MPIs per node with 1 GPU per MPI (right)

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.

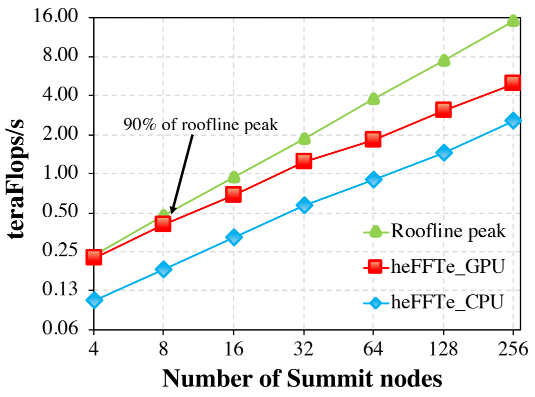

Fig. 8. Roofline performance from Eq. 3 and *heFFTe* performance on a 3D FFT of size 1024^3; using 40 MPI processes, 1MPI/core, per node (blue), and 6 MPI/node, 1MPI/1GPU-Volta100, per node (red).

4.2 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

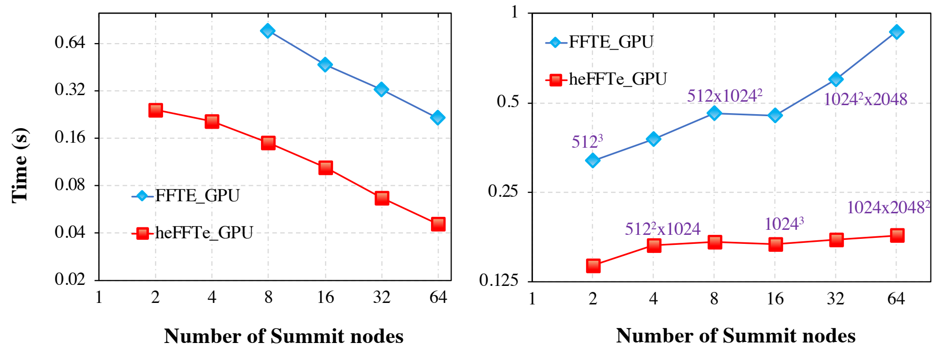

Fig. 7. Strong scalability for a 1024^3 FFT (left), and weak scalability comparison (right). Using 40 MPI processes, 1MPI/core, per node (blue), and 6 MPI processes with 1MPI/GPU-Volta100 per node (red).

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.

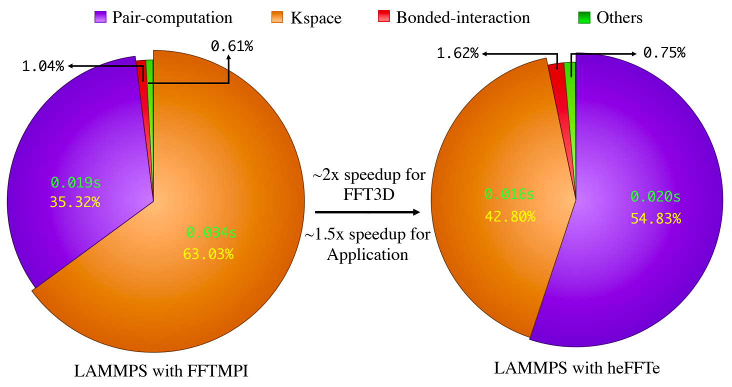

Fig. 9. LAMMPS Rhodopsin protein benchmark on a 128^3 FFT grid, using 2 nodes, 4 MPI processes per node. For FFTMPI we use 1 MPI per core plus 16 OpenMP threads, and for *heFFTe* we use 1 MPI per GPU.

In Fig. 9 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.

4, it is expected that even for large runs, using LAMMPS with *heFFTe* would provide a 2× speedup of its KSPACE routine.

5 Conclusions

In this article, we presented the methodology, implementation and performance results of *heFFTe* library, which performs FFT computation on heterogeneous systems targeting exascale. We have provided experiments showing considerable speedups compared to state-of-the-art libraries, and that linear scalability is achievable. We have greatly speedup local kernels getting very close to the experimental roofline peak on Summit (a large heterogeneous cluster which ranks first on the top 500 supercomputers). Our results show that further optimizations would require better hardware interconnection and/or new communication-avoiding algorithmic approaches.

References

1. `https://github.com/clMathLibraries/clFFT.`
2. `https://bitbucket.org/icl/heffte.`
3. `https://github.com/NVIDIA/nccl.`
4. Ayala, A., Luo, Xi, T.S., Shaiek, H., Haidar, A., Bosilca, G. Dongarra, J.: Impacts of Multi-GPU MPI Collective Communications on Large FFT Computation. In: 2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI) (2019)
5. Czechowski, K., McClanahan, C., Battaglino, C., Iyer, K., Yeung, P.K., Vuduc, R.: On the communication complexity of 3D FFTs and its implications for exascale (06 2012). https://doi.org/10.1145/2304576.2304604
6. Demmel, J.: Communication-avoiding algorithms for linear algebra and beyond. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (2013)
7. Emberson, J., Frontiere, N., Habib, S., Heitmann, K., Pope, A., Rangel, E.: Arrival of First Summit Nodes: HACC Testing on Phase I System. Tech. Rep. MS ECP-ADSE01-40/ExaSky, Exascale Computing Project (ECP) (2018)
8. Filippone, S.: The ibm parallel engineering and scientific subroutine library. In: Dongarra, J., Madsen, K., Waśniewski, J. (eds.) Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science. pp. 199–206. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
9. Franchetti, F., Spampinato, D., Kulkarni, A., Popovici, D.T., Low, T.M., Franusich, M., Canning, A., McCorquodale, P., Van Straalen, B., Colella, P.: FFTX and SpectralPack: A First Look. IEEE International Conference on High Performance Computing, Data, and Analytics (2018)
10. Frigo, M., Johnson, S.G.: 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. Gholami, A., Hill, J., Malhotra, D., Biros, G.: Accfft: A library for distributed-memory FFT on CPU and GPU architectures. CoRR **abs/1506.07933** (2015), `http://arxiv.org/abs/1506.07933`
12. Grama, A., Gupta, A., Karypis, G., Kumar, V.: 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/, https://software.intel.com/en-us/mkl/features/fft
14. Large-scale atomic/molecular massively parallel simulator (2018), available at https://lammps.sandia.gov/
15. Lee, M., Malaya, N., Moser, R.D.: 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. Lin, S., Liu, N., Nazemi, M., Li, H., Ding, C., Wang, Y., Pedram, M.: Fft-based deep learning deployment in embedded systems. In: 2018 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 1045–1050 (2018)
17. Nukada, A., Sato, K., Matsuoka, S.: Scalable multi-GPU 3-D FFT for TSUBAME 2.0 supercomputer. High Performance Computing, Networking, Storage and Analysis (2012)
18. NVIDIA, C.: CUFFT library (2018), http://docs.nvidia.com/cuda/cufft
19. Pekurovsky, D.: P3dfft: A framework for parallel computations of fourier transforms in three dimensions. SIAM Journal on Scientific Computing **34**(4), C192–C209 (2012). https://doi.org/10.1137/11082748X, https://doi.org/10.1137/11082748X
20. Plimpton, S., Kohlmeyer, A., Coffman, P., Blood, P.: fftMPI, a library for performing 2d and 3d FFTs in parallel. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2018)
21. Plimpton, S.J.: FFTs for (mostly) particle codes within the doe exascale computing project, 2017
22. Popovici, T., Low, T.M., Franchetti, F.: Large bandwidth-efficient FFTs on multicore and multi-socket systems. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE (2018)
23. Richards, D., Aziz, O., Cook, J., Finkel, H., et al.: Quantitative performance assessment of proxy apps and parents. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States) (2018)
24. Shaiek, H., Tomov, S., Ayala, A., Haidar, A., Dongarra, J.: GPUDirect MPI Communications and Optimizations to Accelerate FFTs on Exascale Systems. Extended Abstract icl-ut-19-06 (2019-09 2019)
25. Takahashi, D.: 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)
26. Takahashi, D.: FFTE: A fast Fourier transform package. http://www.ffte.jp/ (2005)
27. Tomov, S., Haidar, A., Ayala, A., Schultz, D., Dongarra, J.: 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. Tomov, S., Haidar, A., Ayala, A., Shaiek, H., Dongarra, J.: FFT-ECP Implementation Optimizations and Features Phase. Tech. Rep. ICL-UT-19-12 (2019-10 2019)
29. Tomov, S., Haidar, A., Schultz, D., Dongarra, J.: 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