A More Portable HeFFTe: Implementing a Fallback Algorithm for Scalable Fourier Transforms

Daniel Sharp
Center for Computational Science & Engineering
Massachussetts Institute of Technology
Cambridge, Massachussetts 02139
dannys4@mit.edu

Miroslav Stoyanov
Multiscale Methods Group
Oak Ridge National Laboratory
Oak Ridge, TN 37831
stoyanovmk@ornl.gov

Stanimire Tomov
Innovative Computing Laboratory
University of Tennessee
Knoxville, TN 37996
tomov@icl.utk.edu

Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
Knoxville, TN 37996
dongarra@icl.utk.edu

***Abstract*—The Highly Efficient Fast Fourier Transform for Exascale (heFFTe) numerical library is a C++ implementation of distributed multidimensional FFTs targeting heterogeneous and scalable systems. To date, the library has relied on users to provide at least one installation from a selection of well-known libraries for the single node/MPI-rank one-dimensional FFT calculations that heFFTe is built on. In this paper, we describe the development of a CPU-based backend to heFFTe as a reference, or "stock", implementation. This allows the user to install and run heFFTe without any external dependencies that may include restrictive licensing or mandate specific hardware. Furthermore, this stock backend was implemented to take advantage of SIMD capabilities on the modern CPU, and includes both a custom vectorized complex data-type and a run-time generated call-graph for selecting which specific FFT algorithm to call. The performance of this backend greatly increases when vectorized instructions are available and, when vectorized, it provides reasonable scalability in both performance and accuracy compared to an alternative CPU-based FFT backend. In particular, we illustrate a highly-performant $\mathcal{O}(N \log N)$ code that is about $10\times$ faster compared to non-vectorized code for the complex arithmetic, and a scalability that matches heFFTe's scalability when used with vendor or other highly-optimized 1D FFT backends. The same technology can be used to derive other Fourier-related transformations that may be even not available in vendor libraries, e.g., the discrete sine (DST) or cosine (DCT) transforms, as well as their extension to multiple dimensions and $\mathcal{O}(N \log N)$ timing.**

I. Introduction

The Fourier transform is renowned for its utility in innumerable problems in physics, partial differential equations, signal processing, systems modeling, and artificial intelligence among many other fields [1], [2]. The transform can be represented as an infinite dimensional linear operator on the Hilbert space of "sufficiently smooth" functions, but becomes a finite dimensional linear operator when applied on the space of functions with compact support in the frequency domain [2]. Since any finite-dimensional linear operator can be represented as a matrix, this transformation is equivalent to a "discrete" Fourier transform (DFT) requiring $\mathcal{O}(N^2)$ operations on a signal with N samples, bounding the performance of a DFT from above. However, it is commonly taught that the transform can be accelerated and computed in $\mathcal{O}(N \log N)$ operations using the "Fast Fourier Transform" (FFT), a class of algorithms pioneered in the late 20th century [3]–[5].

Currently, the landscape for computing the one-dimensional FFT of a signal on one node includes many respectable implementations, including those of Intel's OneMKL initiative, NVIDIA's cuFFT, AMD's rocFFT, and FFTW [6]–[9]. This list includes implementations for both CPU and GPU devices, largely giving flexibility to a user needing to compute the FFT of a few small signals on a local machine, a few intermediate-sized signals on a robust compute device, or perhaps many independent small- and intermediate-sized signals on a larger, heterogeneous machine. However, these libraries are seldom designed for the problem of scale— as scientists desire the frequency representation of increasingly large multidimensional signals, they will at some point need to shift towards using distributed and heterogeneous machines. Creating scalable FFTs for large peta- or exascale distributed machines is an open problem, and the heFFTe [10] library has the ambition to be the most performant on this frontier.

Up to this point, the heFFTe [11] library has been fully dependent on the aforementioned one-dimensional FFT packages, requiring the user to install and link to external dependencies for both testing and production runs. Some of these libraries require abiding by non-permissive licensing agreements (e.g., FFTW) or proprietary restrictions (e.g., MKL), limiting the use of heFFTe in more sensitive or proprietary domains. Other packages require specialized hardware, e.g., a specific brand's GPU device, and even if such hardware is available on many production machines it is seldom available on the testing environments. These were prime motivations for having some fallback or reference implementation self-contained in heFFTe that was under the full jurisdiction of the

maintainers. Due to the distributed nature of the library, the speed of the algorithm is less critical compared to traditional one-dimensional FFT implementations, as the algorithm is communication and not computation bound. Therefore, the reference backend of the library stresses accuracy first with a secondary focus on speed.

This reference implementation, or "stock FFT", is not just a naïve implementation of the DFT. The fast $\mathcal{O}(N \log N)$ algorithms are employed, and the CPU Single-Instruction Multiple-Data (SIMD) paradigm is used for complex arithmetic. The "stock FFT" implementation also works on batches of data, transforming multiple identically-sized signals at the same time which is the primary use case within the heFFTe framework.

II. Vectorization of Complex Numbers

Many default packages providing complex multiplication, like `std::complex` from the C++ standard library or `complex` from Python, are developed for consistency and compatibility and, thus, will implement complex multiplication as the textbook definition. Given $a, b, c, d \in \mathbb{R}$, the simplest way of performing complex multiplication is via the direct evaluation of $(a + bi)(c + di) = (ac - bd) + (ad + bc)i$. This is generally optimal in terms of floating point operations (flops), where one complex multiplication is four floating point multiplications and two floating point additions, or six flops. However, one must note that a computer performs *instructions*, not flops.

Vectorization has been supported to some degree within high-performing CPUs since the 1970s, and the more modern SSE and AVX instruction sets [12], [13] have exponentially increased the possibilities for accelerating code via extended registers [14]. In most scenarios, vectorization is implemented at the assembly instruction level, and a programmer can interface with the assembly using instrinsics or wrappers in a low level language (e.g., C, C++, FORTRAN); higher-level interfaces also exist and many scientific computing packages use vectorization internally. Examples of vectorized instructions in AVX include basic arithmetic operations, such as element-wise adding, subtracting, multiplying, dividing, and fused multiply-add. Non-arithmetic instructions can range from simple operations, such as permuting the order items in a vector, to complicated ideas, such as performing one step of AES encryption [13]. Many software libraries take advantage of vectorization and as well as other SIMD capabilities of computers for numerical computation, and even FFT calculation [15], [16].

The CPU executes code in terms of instructions, thus it is more natural to represent an algorithm as a set of vector operations as opposed to working with individual numbers. Let $\mathbf{x} = a + bi$ and $\mathbf{y} = c + di$ and consider the product of the two complex numbers:

$$\begin{aligned}\mathbf{x} \times \mathbf{y} &= \begin{bmatrix} a \\ b \end{bmatrix} \times \begin{bmatrix} c \\ d \end{bmatrix} \\ &= \begin{bmatrix} ac - bd \\ ad + bc \end{bmatrix} \\ &= \begin{bmatrix} ac \\ bc \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} bd \\ ad \end{bmatrix} \\ &= \begin{bmatrix} a \\ b \end{bmatrix} \odot \begin{bmatrix} c \\ c \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \left(\begin{bmatrix} b \\ a \end{bmatrix} \odot \begin{bmatrix} d \\ d \end{bmatrix} \right) \\ &= \mathbf{x} \odot \left(\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \mathbf{y} \right) + \\ &\quad \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \left(\left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \mathbf{x} \right) \odot \left(\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \mathbf{y} \right) \right)\end{aligned}$$

where $\odot$ represents the Hadamard product (i.e. elementwise multiplication). Each operation on individual vectors can be done in one vectorized instruction and, accounting for the capabilities of Fused-Multiply Add, complex multiplication can then be done in five vector instructions, with three of those being shuffle operations that are much cheaper than flops [13].

The advantage of the vectorization is further magnified when multiplying many complex numbers. For example, if

$$\mathbf{x} = \begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} c_1 & c_2 \\ d_1 & d_2 \end{bmatrix},$$

and we want to do the column-wise multiplication of $\mathbf{x}$ and $\mathbf{y}$ (i.e. find $(a_1, b_1) \times (c_1, d_1)$ and $(a_2, b_2) \times (c_2, d_2)$), then we can use the same set of 5 operations but with wider registers, e.g., 256-bit AVX as opposed to 128-bit SSE. Using AVX registers and single precision, we can multiply four pairs of complex numbers in five instructions instead of doing 24 individual flops. Further, CPUs equipped with AVX-512 instructions can execute this complex multiplication on eight pairs of single-precision complex numbers and maintain five instructions.

High level programming languages, such as C and C++, rely on the compiler to convert simple floating point operations into vector instructions, which works well in the simpler instances. However, the shuffle operations used in complex arithmetic are presenting too much of challenge for the commonly used compilers, e.g., see Figure 4. This is despite nearly every general purpose CPU since 2010 supporting some degree of vector instructions and nearly all compute clusters (high-performance or otherwise) supporting these instructions extensively.

The heFFTe library currently allows the user to enable AVX abilities at compile-time and employs them in its stock backend to do all complex arithmetic. The user can also enable AVX512-based complex arithmetic to further increase the library's abilities. These options tremendously increase arithmetic throughput in practice, as seen in Figure 1.

Figure 1 shows that performing arithmetic operations in batches can accelerate a complex algorithm by a significant margin. Of course, this necessitates an algorithm that can take advantage of SIMD, where the instructions are independent of the data.

III. Fast Fourier Transforms

It is worth remarking that, once a matrix is known, all operations of a matrix-vector multiplication are known. The process of evaluating a linear operator is described independent of the data used as an input. Similarly, since a DFT is a finite-dimensional linear operator, all the arithmetic operations are

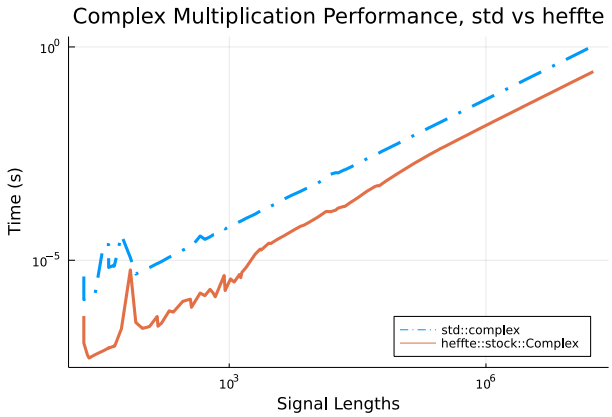

Fig. 1. Creating two sets of eight length-N complex vectors and timing the elementwise multiplication between the sets while scaling N to compare `std::complex` and `heFFTe::stock::Complex`, using `gcc-7.3.0` with optimization flag `O3` in single-precision.

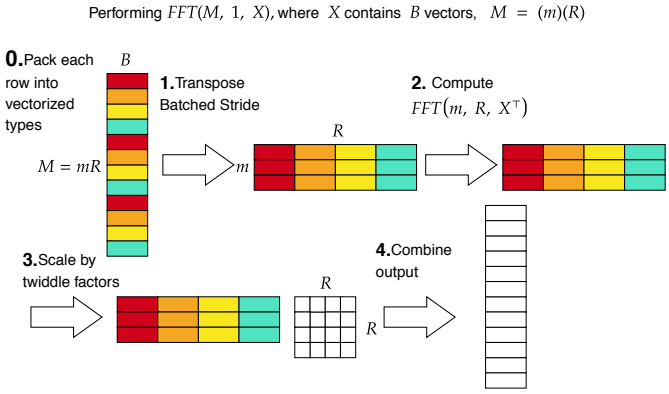

Fig. 2. Example of Cooley-Tukey in heFFTe.

fully determined independent of input content. As such, we can use the idea of vectorized complex numbers to perform one-dimensional FFTs in batches. Since an FFT is fully determined by the size N, two vectors of identical size will have the same sequence of operations regardless of data they contain. As such, if we want the DFT of one single-precision signal, we can get the DFT of up to three more signals in the same number of instructions and similar time when using AVX instructions. The heFFTe library's stock backend enables and encourages this style of batching.

The Cooley-Tukey algorithm [3] forms the foundation for computing FFTs of generic composite-length signals, batched and packed for generic vectorized computing of the FFT of many signals, visualized in Figure 2. Assuming that the user needs to compute P FFTs of length $M = mR$, heFFTe splits this up into batches of size B (depending on the vectorization supported by the machine), then calls the FFTs as illustrated on each batch until all P signals have been transformed.

However, the backend also includes specialized FFTs implemented to calculate signals for length $M = p^\ell$ where p is 2, 3 as well as an implementation of Rader's algorithm [4]

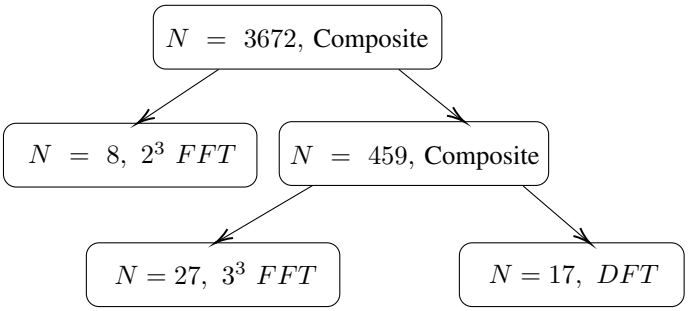

Fig. 3. An Example Call-Graph in the heFFTe Stock Backend.

to calculate the FFT for prime-length signals. Further, the dimensions of $X^\top$ in step 1 of Figure 2 affects the speed of execution. To attempt the fastest FFT, the backend establishes a call-graph of which class of FFT to call recursively and what factors to use a priori. The fact that these call-graphs are created ahead-of-time allows the backend to cache factorization results and other information that might be costly to calculate several times over, thus alleviating some of the computational burden. Additionally, there are optimized FFT implementations for when $N = p^\ell$ for $p = 2, 3$ and when N is prime [3], [4]. An example call-graph is illustrated in Figure 3.

A. heFFTe Integration

The heFFTe library takes as input a distributed signal spread across multiple computer nodes, then uses a series of reshape operations (implemented using MPI) and converts the distributed problem into a series of batch 1D FFT transforms. The user then selects a backend library from a collection to handle the 1D transforms, and the native stock option is part of that collection. However, unlike any of the other libraries, this comes prepackaged with heFFTe so the library will be usable without external dependencies. The stock backend is implemented in C++-11 and the use of AVX vectorization is optionally enabled at compile time, since not all devices can support the extended register. If AVX is not enabled, the C++ standard `std::complex` implementation will be used. Additionally, an option is provided so the user can force enable vectorization, e.g., when cross-compiling on a machine without AVX.

B. Implementation and Performance

The heFFTe library distributes the work associated with FFT via the MPI standard, similar to prior work on distributed and heterogeneous FFT libraries [17]–[20]. Each MPI rank of heFFTe is tasked with performing a set of one-dimensional Fourier transforms. The new integration is built to take a set of one-dimensional signals, package them in the vectorized complex type, perform an FFT (in batches), then unload the vectorized outputs into `std::complex` for communication across the ranks. The backend additionally uses the precision

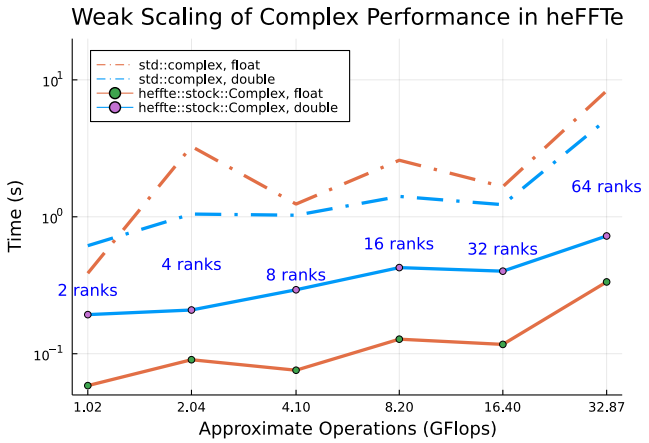

Fig. 4. Performance of the heFFTe library using the stock backend with `std::complex` and `heffte::stock::Complex` numbers, single-precision.

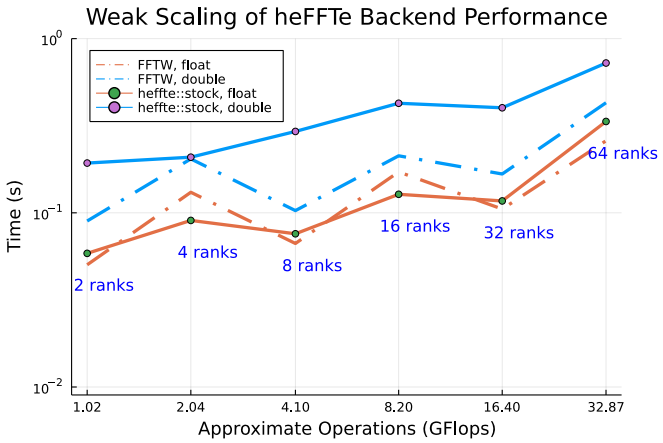

Fig. 5. Benchmarking the stock backend versus FFTW for Complex-to-complex transforms on single- and double-precision signals

and architecture as information to batch in the largest implemented size the CPU can handle (e.g., batches of two for double precision, and four for single precision using 256-bit AVX).

Figure 4 shows a near tenfold increase in performance in heFFTe when using the vectorized complex numbers and a realistic benchmark[1]. This shows that, all else being equal, the vectorized arithmetic's acceleration propagates through an entire call stack instead of being exclusive to some pathological benchmark.

We compare performance results against FFTW [6], which is the most comparable to our implementation. Both FFTW and the stock backend allow for the user to employ AVX512 for performing the FFTs with SIMD. Figure 5 shows that the both the stock and FFTW backends for heFFTe are competitive in many cases, especially regarding single-precision. The FFTW library is mature and extensively optimized with better support for the given CPU's architecture. Additionally, the stock backend scales at the same rate as FFTW, so any future optimizations will most likely be minimizing overhead of the current library, as opposed to making substantial changes to structure of the backend.

The error of this fallback implementation is shown in Figure 6, which demonstrates that the error is as dependent on the problem size as the performance. The single-precision transform seems to generally be between one and three orders of magnitude of error, where the double-precision is generally around one to two orders of magnitude of error. This error is likely attributable to a reasonable amount of floating-point rounding error accumulated while calculating twiddle factors in the transform.

When examining the behavior while strong scaling on a box with a power of four axis size in Figure 7, the stock backend shows a consistent match, if not improvement, in performance

[1]All weak scaling performance was examined on cubes with side lengths of 128, 159, 198, 246, 306, and 381 on a Intel(R) Xeon(R) Gold 6140 CPU equipped with AVX-512

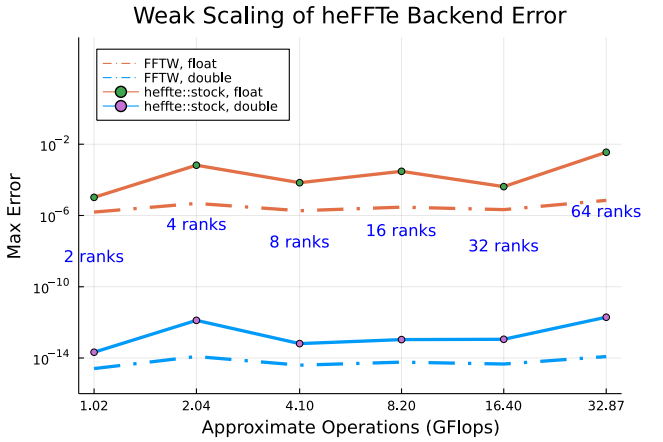

Fig. 6. Error of the Complex-to-Complex transform using the Stock and FFTW backends on single- and double-precision signals

compared to FFTW. On lower rank counts, the single-precision implementation consistently seems to outperform FFTW. As one would expect, the two backends seem to converge to the same elapsed time as the ranks increase and the communication overhead becomes larger than the time to perform each transform.

IV. Conclusions and Future Work

Creating a fallback set of FFT implementations has shown reasonable performance within heFFTe, and incorporating vectorized types accelerates the arithmetic and implementations immensely. Adding a native backend to the heFFTe software package with sufficient performance for most problems allows users the flexibility, e.g., for testing, continuous integration and even small scale production runs. Further, the unrestrictive licensing that heFFTe provides makes it viable to incorporate the library with the stock backend into most projects, regardless of propriety or topic sensitivity. This fallback implementation is included and documented within the development version of heFFTe and will be included in the forthcoming full release version. There are many definitive avenues for the growth

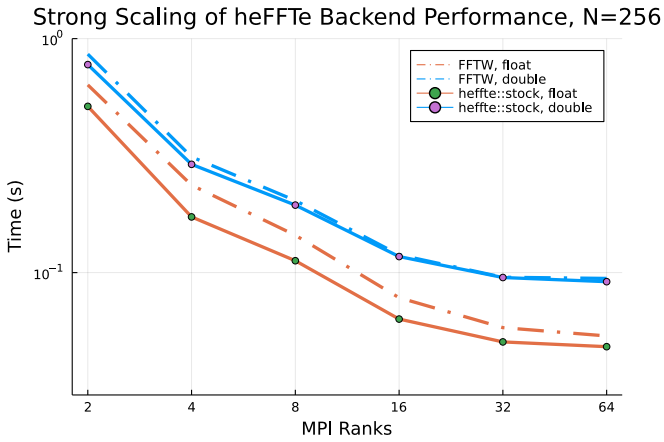

Fig. 7. Performance of the FFTW and stock backends for a fixed- sized signal over multiple MPI ranks, single-precision

and acceleration of this backend; extending support to ARM vectorized instructions would prepare the backend for the heterogeneity of high-performance computing. Other avenues for growth include testing other vectorizations for complex arithmetic and using more specialized algorithms for common, but specific, problem sizes. For example, accelerating the transform on prime-lengthed signals. Further, the error should be reduced, which requires adjusting how twiddle factors are created in the stock implementation. Overall, this is an initial step towards allowing users of the heFFTe library further flexibility in how they use the library and what projects they can use it in.

Future work includes further optimizations and extensions to other architectures, e.g., GPUs from Nvidia, AMD, and Intel, as well as other algorithms. Of particular interest is to show that the same technology can be used to derive other Fourier-related transformations that are highly needed but not always available in vendor libraries, e.g., the discrete sine (DST) or cosine (DCT) transforms, as well as their extension to multiple dimensions and $\mathcal{O}(N \log N)$ timing.

Acknowledgment

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.

References

[1] L. R. Rabiner and B. Gold, *Theory and application of digital signal processing / Lawrence R. Rabiner, Bernard Gold.* Englewood Cliffs, N.J.: Prentice-Hall, 1975.

[2] R. N. Bracewell, *Fourier Transform and its Applications*. McGraw-Hill, 1999.

[3] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[4] C. M. Rader, "Discrete Fourier transforms when the number of data samples is prime," *Proceedings of the IEEE*, vol. 56, no. 6, pp. 1107–1108, 1968.

[5] L. Bluestein, "A linear filtering approach to the computation of discrete fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 18, no. 4, pp. 451–455, 1970.

[6] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[7] "cuFFT library," 2018. [Online]. Available: http://docs.nvidia.com/cuda/cufft

[8] "rocFFT library," 2021. [Online]. Available: https://github.com/ROCmSoftwarePlatform/rocFFT

[9] Intel, "Intel Math Kernel Library," http://software.intel.com/en-us/articles/intel-mkl/. [Online]. Available: https://software.intel.com/mkl/features/fft

[10] "heFFTe library," 2020. [Online]. Available: https://bitbucket.org/icl/heffte

[11] A. Ayala, S. Tomov, A. Haidar, and J. Dongarra, "heFFTe: Highly Efficient FFT for Exascale," in *ICCS 2020. Lecture Notes in Computer Science*, 2020.

[12] "Amd64 architecture programmer's manual, volume 3: General-purpose and system instructions," 10 2020. [Online]. Available: https://www.amd.com/system/files/TechDocs/40332.pdf

[13] "Intel SSE and AVX Intrinsics," 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/

[14] R. Espasa, M. Valero, and J. E. Smith, "Vector architectures: past, present and future," in *Proceedings of the 12th international conference on Supercomputing*, 1998, pp. 425–432.

[15] M. K. Stoyanov and USDOE, "HALA: Handy Accelerated Linear Algebra," 11 2019. [Online]. Available: https://www.osti.gov//servlets/purl/1630728

[16] D. McFarlin, F. Franchetti, and M. Püschel, "Automatic Generation of Vectorized Fast Fourier Transform Libraries for the Larrabee and AVX Instruction Set Extension," in *High Performance Extreme Computing (HPEC)*, 2009.

[17] H. Shaiek, S. Tomov, A. Ayala, A. Haidar, and J. Dongarra, "GPUDirect MPI Communications and Optimizations to Accelerate FFTs on Exascale Systems," University of Tennessee, Knoxville, Extended Abstract icl-ut-19-06, 2019-09 2019.

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

[19] S. Plimpton, A. Kohlmeyer, P. Coffman, and P. Blood, "fftMPI, a library for performing 2d and 3d FFTs in parallel," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2018.

[20] J. L. Träff and A. Rougier, "MPI collectives and datatypes for hierarchical all-to-all communication," in *Proceedings of the 21st European MPI Users' Group Meeting*, 2014, pp. 27–32.