High Performance Kernels for FFT via Modern C++

Paul Caprioli
Robby Jenkins
support@hpkfft.com

ABSTRACT

We present a library for computing the Fast Fourier Transform (FFT) with an interface that fully embraces the principles of modern C++. We support half, single, and double precision; in-place and out-of-place transforms; scaling; arbitrary dimensions; and both complex and real time domain data. We currently support AVX2 and AVX512 hardware.

The API of `hpk::fft` is based on the abstract factory design pattern. We have different types for in-place and out-of-place and template on both precision and domain types to allow static type checking at compilation time. Scaled and unscaled functions have distinct names, and the scale factor is provided at each call site, not beforehand. FFT compute objects are immutable, thread-safe, fully initialized at construction, and managed by smart pointers.

Our results are overall more accurate than the leading vendor library. For half precision, only 58 FFT lengths are vendor supported, and on those, we have slightly lower error: a little better than 3% using the geometric mean. Our error for AVX512 single precision is 15% lower in the mean; for AVX512 double precision, 25% lower.

Furthermore, our performance is generally higher. In AVX512 half precision, our geometric mean performance on the vendor supported sizes is 15% higher. That grows to 20% when evaluated using a small batch. For single and double precision, our AVX512 performance is over 30% higher in the mean. For AVX2, it is over 40% higher.

1 BACKGROUND

The Discrete Fourier Transform (DFT) transforms a sequence of N complex numbers in the time domain

$$\mathbf{x} = \langle x_0, x_1, \ldots, x_{N-1} \rangle$$

into another sequence of N complex numbers in the frequency domain

$$\mathcal{F}(\mathbf{x}) = \mathbf{X} = \langle X_0, X_1, \ldots, X_{N-1} \rangle$$

such that

$$X_k = \alpha \sum_{n=0}^{N-1} x_n \exp\left(\frac{-2\pi i}{N} kn\right)$$

where α is a scaling factor and $i = \sqrt{-1}$.

The backward DFT goes the other way, transforming a sequence of complex numbers in the frequency domain into another sequence in the time domain

$$\mathcal{F}^{-1}(\mathbf{X}) = \mathbf{x} = \langle x_0, x_1, \ldots, x_{N-1} \rangle$$

such that

$$x_k = \beta \sum_{n=0}^{N-1} X_n \exp\left(\frac{+2\pi i}{N} kn\right)$$

where β is the backward scaling factor.

If the two scaling factors are chosen such that $\alpha \cdot \beta \cdot N = 1$, then the forward and backward DFTs are inverse functions of each other. That is,

$$\mathbf{x} = \mathcal{F}^{-1}\big(\mathcal{F}(\mathbf{x})\big) \quad \text{and} \quad \mathbf{X} = \mathcal{F}\big(\mathcal{F}^{-1}(\mathbf{X})\big).$$

So, common choices are $\alpha = 1$, $\beta = 1/N$ and $\alpha = \beta = 1/\sqrt{N}$, though the scale factor is sometimes selected either statically or dynamically for the practical purpose of preventing overflow. For example, when using half precision (IEEE Std. 754–2019 binary16) floating point for processing signals quantized by 16-bit analog-to-digital converters, the constant $1/256$ may be a good choice [21].

The Fast Fourier Transform (FFT) is an algorithm for computing the DFT that has log-linear complexity, i.e., $O(N \log N)$. This speedup (vs. the $O(N^2)$ complexity of a straightforward matrix-vector product) has revolutionized numerical computing [7] [12], with the pivotal moment being the 1965 publication of the algorithm by Cooley and Tukey that now bears their names [8]. Despite the seeming simplicity of the mathematics, books have been written on its implementation and applications [5], and its relevance to exascale-level supercomputing is indisputable [3].

For example, VASP has been the top workload at the U.S. National Energy Research Scientific Computing center [2] and at the U.K. national supercomputing service [22]. Typically, molecular dynamics codes such as VASP and GROMACS use the Particle Mesh Ewald method to efficiently calculate long range electrostatics, the speed of which is significantly dependent on 3D FFT calculations [1].

On the petascale-level supercomputer Blue Waters at the National Center for Supercomputing Applications, 16% of total node-hours were attributed to computing FFTs, which is higher than the 13% for dense matrix calculations. Approximately 80M node-hours were spent in the FFTW library alone. The science areas using FFTs include molecular dynamics (AMBER, GROMACS, LAMMPS, NAMD), quantum chromodynamics (Chroma, CPS, MILC), material science (CP2K, CPMD, Quantum Espresso, VASP), turbulence (DISTUF, MIRANDA, PSDNS), fluid dynamics (Fluent, NEK5000), plasma and magnetosphere simulation (VPIC), and general relativity (SpEC, Spectre). The top application was NAMD, used mainly for molecular bioscience, which consumed about 18% of all machine cycles. The next three applications were Chroma and MILC for physics and AMBER for chemistry. Note that all of the top four applications use the FFT [16].

2 PRINCIPLES OF LIBRARY DESIGN

In this section, we explain the architectural design of `hpk::fft` at a higher level and share software engineering advice with the hope of influencing supercomputing libraries beyond FFTs. We hold that a library is best designed from the perspective of the developers who will be using it, despite occasional inconvenience to the developers of the library itself.

2.1 Avoid Globals

The `hpk::fft` library has no global settings, and it does not define its own environment variables. So, for example, single and multi-threaded compute objects can coexist in an application. Also, there is no global cache or memory pool, and there are no functions that manage global state, such as `free_all_buffers()`.

2.2 Use Common Idioms

A library intended for C++ developers should embrace its idioms. The interface needs to match the programmer's way of thinking; it is not enough that a compiler is able to parse its header files. It's also a good idea to follow the most widely agreed-upon stylistic conventions, e.g., writing macro names but not enumerators in all upper case [20, Enum.5].

A fundamental idiom in C++ is RAII, or "Resource Acquisition Is Initialization." Resources (such as memory) are acquired when an object is constructed and released when it is destroyed. When the last `hpk` object is destroyed (e.g., its lifetime ends by going out of scope), all resources are released.

Another rule is that construction should create fully initialized objects so that users may assume that a constructed object is usable [20, C.41]. For example, `hpk::fft` objects that compute FFTs are constructed and initialized by a single function call. The objects are never in a partially initialized state. In fact, they are immutable.

Moreover, FFT objects are owned and managed by smart pointers. As observed by Scott Meyers [18, Ch. 4]:

> Raw pointers are powerful tools, to be sure, but decades of experience have demonstrated that with only the slightest lapse in concentration or discipline, these tools can turn on their ostensible masters.... You should therefore prefer smart pointers to raw pointers....
> A common use for `std::unique_ptr` is as a factory function return type for objects in a hierarchy.

By returning a `std::unique_ptr`, the object is easily and efficiently passed to other application functions (either by moving or by converting to a `std::shared_ptr` and copying), and it is automatically disposed of when the owning pointers go out of scope.

2.3 Avoid Limitations

The fewer arbitrary limits, the better. For example, `hpk::fft` supports FFTs of any length, not just powers of two or products of small primes. The scale factor is provided as a real value, not as an enumeration (such as `Scale::div_by_sqrtn`). There is no predefined upper bound on the number of dimensions in a multidimensional FFT. Floating point precision (half, single, double) is orthogonal to other library features. The time domain can be real or complex.

Striding is supported, and strides are always measured in terms of real elements. For example, adjacent complex points have stride two. Consider the following:

```
struct Datum {
    int id;
    std::complex<double> x;
};
```

or, equivalently,

```
struct Datum {
    int id;
    double x_real;
    double x_imag;
};
```

In either case, `sizeof(Datum)` is 24 and `alignof(Datum)` is 8. So, in order to be able to compute an FFT on the `x` values of

```
std::vector<Datum> data;
```

one would specify the stride to be 3. A library interface that specifies strides in terms of complex points would effectively limit strides to an even number of doubles and preclude this computation.

`Inplace` objects necessarily apply the same strides to both their input and output. For `Ooplace` objects, strides are individually specified for the time domain and for the frequency domain, not for a function's input and output. This allows a single FFT compute object to perform both a forward transform (having input with time stride and output with freq stride) and also its inverse (a backward transform having input with freq stride and output with time stride).

2.4 Algorithmic Complexity

The complexity properties of a library function are an important part of its user interface, even though it is expressed in documentation rather than in machine readable header files. Since an FFT can be computed in $O(N \log N)$ time for all N via convolution with a chirp filter [4], an FFT library should have log-linear complexity for any input length. In this, the FFTW3 package has set the standard [9].

2.5 Leverage Static Typing

The easiest errors to fix are those reported at compile time. The compiler's error message is usually helpful and at least provides a line of source code at which to begin an investigation. Link errors are harder and are usually the result of misordering or omitting libraries, but they also arise from referencing a symbol that simply does not exist. Runtime errors are by far the most difficult, and this is especially true when the program produces wrong results without crashing.

The structure and hierarchy of types provided by a library is crucial to its usability. Types should be architected to distinguish objects that should be used differently so that static checking can discover bugs at compile time. Contrariwise, objects that are substitutable for one another should have the same type. This can be accomplished in an object-oriented language using an abstract base class or interface type.

Choosing what information to encode in a library's types requires careful thought. In `hpk::fft`, we distinguish FFT compute objects based on the following: floating point precision (half, single, double), the data representation in the time and frequency domains (e.g., complex), and whether the computation is in-place (the output overwrites the input) or out-of-place (the output is disjoint from the input). Forward and backward transforms, as well as scaled and unscaled transforms, are separated into individual functions, with each compute object providing all four.

We do not vary types according to hardware architecture nor, of course, based on internal algorithmic implementation. We follow

the abstract factory pattern [11], in which the factory object's type is itself an abstract class. In particular, the factory's type includes the data representation of the FFT objects it can make (e.g., double precision, complex to complex FFTs) but not anything about the hardware. Different concrete factories (for AVX2, AVX512, and later Armv9, CUDA, etc.) inherit from the same abstract class.

Application code first makes a factory, typically allowing hardware detection to choose among the available concrete types. The client then requests the factory to make, say, an `Inplace` object. This too is an abstract type that includes the data representation, but application code is not burdened by, and is not even aware of, the specific derived concrete type of the object. It's easier than it sounds; in fact, it can all be done in 69 characters:

Listing 1: In-place one line example

```
#include <hpk/fft/makeFactory.hpp>

int main() {
  std::vector<std::complex<double>> v =
          {7.0, 0.0, 0.0, 0.0};
  hpk::fft::makeFactory<double>()
          ->makeInplace({4})
          ->forward(v.data());
}
```

Usually an application will compute more than one FFT. Both the factory and the compute object can be named, as shown in Listing 2. Note that these objects *are* statically typed; the types are deduced by the compiler. Writing `auto` just saves keystrokes. Below, both `factory` and `fft` are `std::unique_ptr`, with the former owning an `hpk::fft::FactoryCC<double>` and the latter owning an `hpk::fft::OoplaceCC<double>`.

Listing 2: Out-of-place example with variables

```
#include <hpk/fft/makeFactory.hpp>

int main() {
  std::vector<std::complex<double>> x =
         {7.0, 0.0, 0.0, 0.0};
  std::vector<std::complex<double>> X(4);
  auto factory =
         hpk::fft::makeFactory<double>();
  auto fft = factory->makeOoplace({4});
  fft->forwardCopy(x.data(), X.data());
}
```

2.6 Function Overloading

Overloading allows functions that have the same semantics to share the same function name, reducing the mental effort of learning and remembering a library interface. To avoid errors and confusion, functions with different semantics should be given different names.

In particular, functions for performing unscaled FFTs calculate a different result than do scaled FFTs, and functions that overwrite their input data have different semantics from those that do not. In `hpk::fft`, the functions for performing an in-place FFT are:

- `forward`
- `backward`
- `scaleForward`
- `scaleBackward`

Those that copy their results to disjoint memory are:

- `forwardCopy`
- `backwardCopy`
- `scaleForwardCopy`
- `scaleBackwardCopy`

It would be needlessly confusing to collapse all of these into only one or two function names. Note, by the way, that the scale factor is provided as an argument to the function at the call site, not prior.

On the other hand, we do make significant use of overloading. For example, while users must agree on the actual data format (i.e., real and imaginary parts interleaved in memory), some may use an array of `float` as their type while others prefer an array of `std::complex<float>`. For this reason, the functions listed above are overloaded to accept a pointer to a real type whenever an input or output argument formally requires a pointer to a complex type. This is done in the header file using templates and so incurs no run-time overhead.

Furthermore, also using templates, these functions accept a final argument that optionally specifies the source of scratch memory necessary for the computation. This argument can be a raw pointer, a smart pointer, or an allocator. If omitted (as in Listing 1 and 2) the allocator overload is selected, with an `hpk::AlignedAllocator` constructed by default.

2.7 Free vs. Member Functions

A class is an assemblage of data and the functions that operate on it. If a function or procedure can be implemented as a non-member, non-friend function, then doing so is the better choice. It enhances encapsulation. Note that member functions and friend functions are logically part of a class as these both have access to its private data.

Alexander Stepanov and Paul McJones [19] write:

> A *computational basis* for a type is a finite set of procedures that enable the construction of any other procedure on the type. A basis is *efficient* if and only if any procedure implemented using it is as efficient as an equivalent procedure written in terms of an alternative basis.

A class not only should provide an efficient basis but also should avoid including member or friend functions that can be implemented equally well via its public interface.

A good example is the BLAS function `gemm`, which performs general matrix multiplication. It is a free function, not a member of some `Matrix` class, and it is not a friend function—the elements of a matrix are efficiently accessed using an existing public interface.

On the other hand, the interface to an FFT is through the compute functions themselves. For efficiency, the complex exponential in the mathematical formulation is evaluated at the necessary points on the unit circle when the FFT object is constructed. These "twiddle

factors" are a private implementation detail, as they depend on algorithmic choices as well as on aspects of the processor's architecture (such as SIMD register width).

Since the FFT compute functions (`forward`, etc.) must access the private internals of `Inplace` or `Ooplace`, these functions are naturally class member functions. They cannot be implemented as non-member, non-friend functions, and members are preferred over friends [6]. It is what users expect.

2.8 Exceptions

Avoiding exceptions is *not* a rule in C++ [20, NR.3], but note that exceptions are not for errors that can be handled locally nor are they for errors that require instant termination after a non-recoverable error. So, we do not see compelling value in throwing exceptions for invalid configurations or unimplemented features. Instead, in such cases, we return an empty `std::unique_ptr` with the thought that developers are better served by a simpler interface.

Memory allocation can throw exceptions (e.g., `std::bad_alloc`), but since the bulk of our allocation is done in header file code with a user-supplied allocator, users have full control. The default allocator is `hpk::AlignedAllocator`, which we provide as a header-only implementation. It can be compiled with or without exceptions depending upon compiler flags, viz. `-fno-exceptions`, so we leave this as a choice to our users.

Finally, note that floating point exceptions (e.g., overflow) are not C++ exceptions, so the FFT compute functions themselves are declared `noexcept`. As some developers prefer to avoid exceptions, and there are good reasons for doing so (e.g., hard real-time systems, limited memory embedded systems), the `hpk::fft` library itself does not throw exceptions.

2.9 Thread Safety

Computing an FFT (e.g., calling `fft->forward`) doesn't change the `fft` object, and, conceptually, calling `factory->makeInplace` or `factory->makeOoplace` doesn't change the `factory` object. Therefore, these member functions are `const`-qualified, so users would naturally assume them to be thread safe [18, Ch. 3]. Indeed, they are correct to do so. These `hpk::fft` objects may be shared among threads, and their member functions may be called without the application having to implement locking or other mechanisms for synchronization.

2.10 Strawman Case Study

In Listing 3, below, we fabricate a hypothetical example to serve as a foil for the `hpk::fft` library interface.

At first glance, we see that this library's `fft` descriptor requires initialization after it has been created but before it can be used, and we are left feeling uncertain as to whether we have done so correctly. Perhaps further initialization is needed even after the call to `commit`, such as allocating scratch memory and calling `setWorkspace`.

Also, we note the incongruity of having `computeForward` be a free function, as the computation surely uses private state in `fft` similarly to the two member functions preceding it. From a productivity standpoint, many programmers like the completion tips offered by an IDE after typing `fft->`, but obviously no such tips are available when starting a new line of code.

Listing 3: Broken strawman example

```
int main() {
  std::vector<std::complex<double>> x =
          {7.0, 0.0, 0.0, 0.0};
  std::vector<std::complex<double>> X(4);
  auto fft =
      std::make_unique<
          Descriptor<Precision::float32,
                     Domain::complex>>(4);
  fft->setValue(Parameter::forwardScale,
                "ortho");
  fft->commit();
  computeForward(*fft, x.data(), X.data());
}
```

Regardless, we try compiling our code and enjoy a brief moment of happiness when that succeeds, but then linking fails. There's an undefined reference to `computeForward`, with template parameters wrapping onto five lines. It turns out free functions are declared for a full Cartesian product of arbitrary types,

```
template<typename descriptor_t,
         typename in_t,
         typename out_t>
void computeForward(descriptor_t& desc,
                    in_t* in,
                    out_t* out);
```

and they're all friends of every descriptor. The descriptor itself is templated by non-type template parameters, and in our case the first is `Precision::float32`, an integer such as 35. So our code compiles even though our descriptor is unsuitable for double precision data and no such function exists in the library. We correct the parameter to `Precision::float64`, and now the code links, and that is much, much worse.

The string `"ortho"` was improperly copied from our python prototype; it should have been `0.5`, but since the binary function `setValue` is implemented as a C-style variadic function [20, F.55], type checking is nonexistent. The string's pointer was reinterpreted as some double value. Lastly, we forgot to set the value of `placement` to `not_inplace` before calling `commit`, and so the compute function's behavior is undefined. Errors of omission are particularly hard to recognize in a code review, and since placement is not part of the `Descriptor`'s type, this error escaped undetected.

Static type checking is an intrinsic feature of C++, but this library departs from the spirit and idiomatic usage of the language. To a manager, one might say the library's programming interface is suboptimal; to a Star Trek fan, "Shaka, when the walls fell" [17].

3 ADVANCED API

3.1 Memory Allocation

The two main uses of memory in FFT computations are for storing trigonometric constants ("twiddle factors") and for temporarily storing intermediate ("scratch") values. Twiddle factors are generated when an `Inplace` or `Ooplace` object is made by a factory, and

scratch memory is used temporarily during the course of an FFT computation.

A custom allocator can be provided for each of these purposes, and these allocators do not have to be the same. For example, the first can allocate memory in a system-specific manner that is optimized for write-once, read-many data that is long living and widely shared among threads. The second can be optimized for data that is written, read, and then discarded.

An `Allocator` can be passed as the optional last function argument to `makeInplace` or `makeOoplace`. This will then be used if needed to allocate memory for the twiddle factors. (Small-sized FFTs may not require a twiddle table.)

An `Allocator` can be passed as the optional last function argument to an FFT compute function. This will then be used to allocate scratch memory if needed. (Small-sized FFTs may not need any scratch memory.) Note that, if there are multiple computations, it is generally advisable to allocate scratch memory once and reuse it for each function call to avoid repeatedly allocating and freeing memory. The member functions `scratchSize` or `scratchSizeBytes` return the amount of memory required, measured in terms of real-valued elements or bytes, respectively.

The function `allocateMemory` allocates memory and returns a `std::unique_ptr` that owns it, so the memory will be freed automatically when the pointer goes out of scope. Even simpler, `allocateScratch` will do the same after first calling `scratchSize` to determine the number of elements to allocate. Each of these convenience functions also optionally takes a custom `Allocator` as a last argument.

Listing 4: Reusing scratch memory

```
#include <hpk/fft/makeFactory.hpp>

int main() {
  std::vector<std::complex<double>> v =
         {7.0, 0.0, 0.0, 0.0};
  auto factory =
         hpk::fft::makeFactory<double>();
  auto fft = factory->makeInplace({4});
  auto scratch = allocateScratch(*fft);
  fft->forward(v.data(), scratch);
  fft->backward(v.data(), scratch);
}
```

3.2 Twiddle Caching

Each factory object maintains a single-entry cache that contains the last twiddle table it generated. If the next function call to `makeInplace` or `makeOoplace` from the same factory requires the same twiddle constants, then the twiddle table is reused. For example, in the following:

```
auto inp = factory->makeInplace({100});
auto oop = factory->makeOoplace({100});
```

both FFT objects will share the same twiddle table.

The factory can be destroyed at any time without affecting the objects it has already made. There is no "spooky action at a distance."

Note that, from the outside, a factory is conceptually an immutable object. All of its member functions are `const`. While this is true from a functionality perspective, it is seen as not quite true once memory usage and performance are considered.

3.3 Multithreading Considerations

`Inplace`, `Ooplace`, and `Factory` objects are thread safe. This is true for the FFT compute objects because they are immutable and for the factory objects because, internally, the twiddle table cache is protected by a mutex.

This is something to consider when deciding whether to share a factory object between threads. The downside of sharing a factory object is that only one thread at a time can make an `Inplace` or `Ooplace` object. If each thread were to make and use its own factory, this would not be the case.

On the other hand, if multiple threads will be using a shared factory to make the same `Inplace` or `Ooplace` objects, then those objects will share the same twiddle table. This is especially nice if the threads share the same CPU hardware cache; the twiddle constants would not be replicated needlessly in the processor's data cache.

In a NUMA system, one could make a factory for each NUMA domain. Then, assuming the same size FFTs are made, all the threads in one NUMA domain would share one twiddle table, but threads in a different domain would have a separate copy of the twiddle constants.

Note that, of course, reusing an existing twiddle table (a "hit" in the factory's twiddle cache) is faster than allocating memory and generating trigonometric constants. Under the right circumstances, this could have a performance smoothening effect: the fastest thread calls `makeInplace` first and computes a bunch of complex exponentials, and then the slower threads call `makeInplace` and find the results are ready to use. The slower threads have less work to do.

4 ACCURACY RESULTS

Accuracy results are obtained using benchFFT [10] with the flag `--accuracy-rounds` set to:

$$\begin{cases} 1{,}000{,}000 & \text{if } 1 \le N < 10 \\ 100{,}000 & \text{if } 10 \le N < 100 \\ 10{,}000 & \text{if } 100 \le N < 1{,}000 \\ 1{,}000 & \text{if } 1{,}000 \le N < 10{,}000 \\ 100 & \text{if } 10{,}000 \le N < 100{,}000 \\ 10 & \text{if } 100{,}000 \le N \end{cases}$$

We report the backwards error of the forward out-of-place computation using the Euclidean distance with the idea that this choice best matches human intuition for those who have a sense of the uncertainty in their time domain input data.

For comparison, we use the Intel Math Kernel Library version 2023.0 [14] for single and double precision and the Intel Integrated Performance Primitives version 2021.7 [13] for half precision. These library versions are packaged for download as part of the oneAPI

BaseKit version 2023.0.0. At the time of this writing, MKL does not support half precision FFTs, and IPP has only limited support. In particular, the functions `ippsDFT{Fwd,Inv}_Direct_CToC_16fc` therein do not support scaling and are limited to the 58 specific transform lengths listed in Table 1. For single and double precision, we use the 400 lengths listed in Table 2.

FFTW3 is written in portable C [9], which is an advantage we do not share since our kernels explicitly invoke either AVX2 or AVX512 instruction intrinsics [15], which are extensions supported by both GCC and Clang. (We use `icpx`, the Intel LLVM compiler.) Thus, we feel obligated to compare ourselves to vendor-optimized libraries, which presumably also have chosen to sacrifice easy portability across architectures in the pursuit of performance.

In all of our graphs, the x-axis is the transform length in strictly increasing order. The results are spaced equally apart, and a tic mark is placed at every power-of-two length.

4.1 Half Precision

In Figure 1, we see that our error, as measured experimentally by benchFFT over the 58 lengths supported by IPP, is only slightly better than IPP's. Taking the geometric mean of the ratios of the computed error at each point gives the value 0.964. That is, overall, we are about 3.6% more accurate.

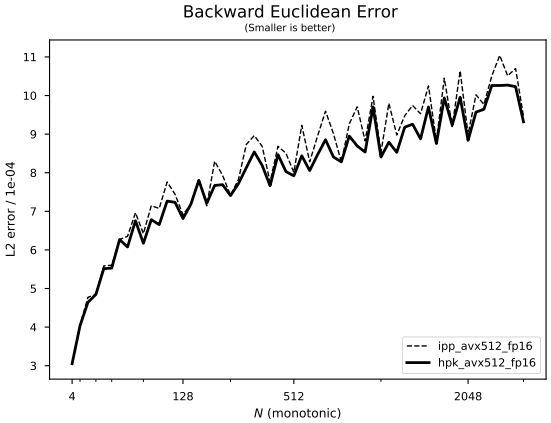

Figure 1: Error for AVX512 half precision

4.2 Single Precision

We see in Figure 2 that for AVX2 our measured error is comparable to MKL's. However, overall, we are slightly more accurate. The geometric mean of the ratios for the 400 lengths is 0.956.

The results for AVX512 are qualitatively different. As seen in Figure 3, `hpk::fft` shows significantly lower error than MKL and, furthermore, is more consistent from one transform length to the next. The geometric mean of the ratios at each N is 0.836.

4.3 Double Precision

For AVX2 in double precision, `hpk::fft` has both lower error than MKL and is more uniform across transform lengths. This is shown in Figure 4. The geometric mean of the ratios is 0.823.

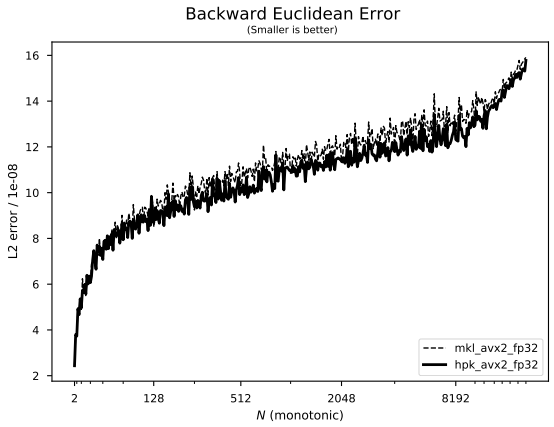

Figure 2: Error for AVX2 single precision

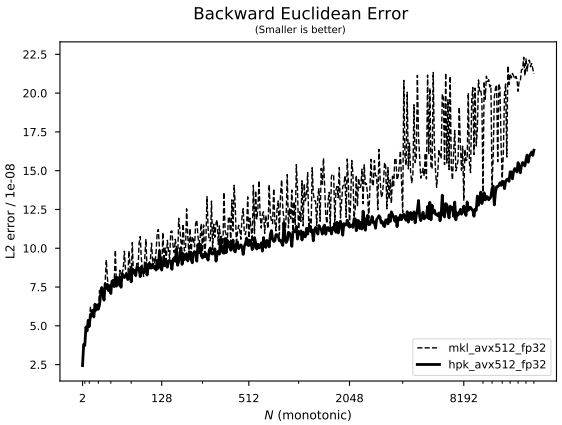

Figure 3: Error for AVX512 single precision

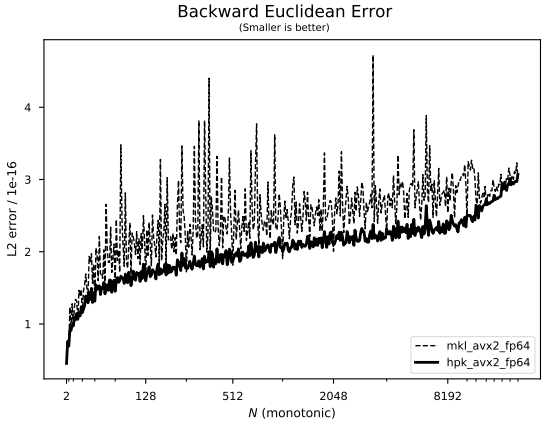

Figure 4: Error for AVX2 double precision

For AVX512 in double precision, as seen in Figure 5, `hpk::fft` is both lower in error as well as more consistent from one length to the next. Overall, the geometric mean of the error ratios is 0.727. Note, however, that MKL's error varies significantly from length to length. For example, for lengths 29400 and 38400, we have half the error (i.e., twice the accuracy) of MKL.

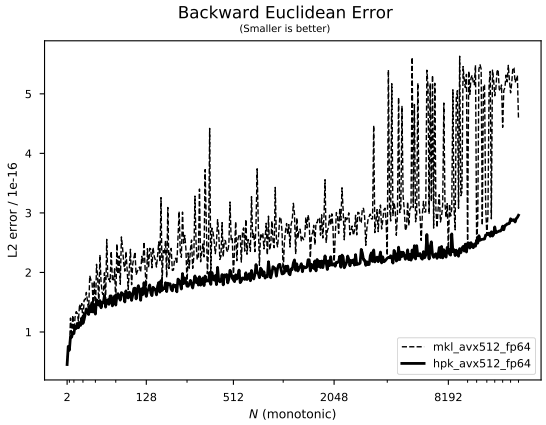

Figure 5: Error for AVX512 double precision

5 PERFORMANCE RESULTS

For performance, the benchFFT [10] flags `-r16` and `-t0.1` were specified. These, respectively, cause each test case to be measured 16 times (reporting the fastest result) and adjust the number of inner loop iterations so that each measurement is for at least 100 ms. We report performance relative to the Intel performance libraries and use the same lengths as in the previous section. These lengths were chosen so as to have a natively supported factorization by both implementations.

For each transform length, four FFTs are computed, and the graphs are created after geomeaning[1] the four performance ratios. For single and double precisions, the four test cases are the cross product of forward and backward, in-place and out-of-place. For half precision, Intel IPP only supports out-of-place computations, so we ran forward and backward twice each. Always having four results made script-writing easier and, with only 58 sizes to benchmark, did not impose an undue computational burden.

All performance results were obtained on a system having two Intel Xeon Platinum 8480+ processors (formerly Sapphire Rapids) with 56 cores per socket and two threads per core. Only single-threaded performance was measured, with this code path selected by setting the environment variable `OMP_NUM_THREADS=1`. We measured AVX2 performance for MKL by setting the environment variable `MKL_ENABLE_INSTRUCTIONS=AVX2` and for `hpk::fft` by making the factory with configuration `hpk::Architecture::avx2`. The operating system was Ubuntu 20.04.5 LTS. Unless otherwise noted, the batch size is one, i.e., a single sequence of length N is transformed.

[1]Having a definition analogous to that of averaging, but for the geometric mean, *geomeaning* is a perfectly cromulent word.

5.1 Half Precision

The implementation of `hpk::fft` relies heavily on C++ templates and generic programming techniques. Consequently, we support half precision in the same way and with the same options as we support single and double precisions. In contrast, IPP supports half precision only for select sizes and has an API for their computation that is distinct from their API for single and double precision. For example, half precision IPP uses static twiddle tables, and users do not (and cannot) supply scratch memory as would otherwise be done by means of `DFTInit`. This limited API allows optimizations to be applied selectively and each length to be individually tuned for performance, and we assume this to be the case. Thus, we are quite pleased to show 15% better performance, as seen in Figure 6, below.

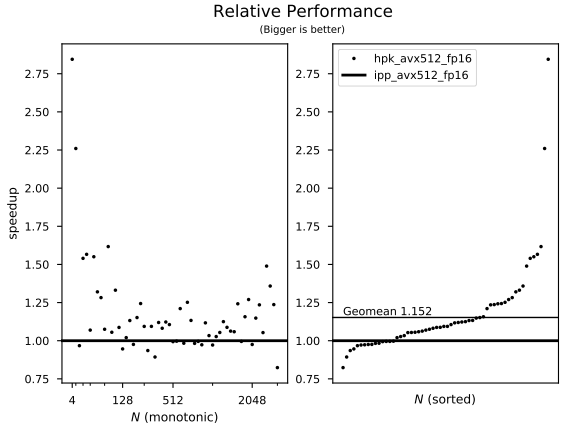

Figure 6: Performance for AVX512 half precision

It is worth mentioning that for a batch size of one, as shown above, we are significantly slower on the 4096-point FFT, having a performance ratio of 0.82. This test case requires 32KiB total for input and output plus whatever is needed for twiddle constants and scratch memory. With care, this can essentially be made to fit in the 48KiB first-level data cache of Sapphire Rapids. We conjecture that doing so accounts for IPP's performance advantage, though we have not attempted to write a freestanding kernel for this size.

In order to get reproducible timing measurements, benchFFT runs each test case in a loop until reaching a minimum run time, which we have set to 100 ms. Note that it requires roughly 1 µs to compute a 4096-point FFT, and so benchFFT runs this computation approximately 100,000 times, repeatedly reading the same input and writing to the same output location. Clearly, fitting in the cache offers ample opportunity for reuse.

We are unsure as to whether this represents a realistic use case. It may, if input and output memory are already in the CPU cache. But, for example, suppose even a small number of 4096-point data sequences need to be transformed. In this case, the input and output data do not fit in the L1 cache, though the twiddle table and scratch memory, which are reused for each transform in the batch, fit easily. Ideally, of course, one has application code and can measure performance in situ, but we are not in that position.

Instead, we do what we can to explore performance while benchmarking with benchFFT. In Figure 7, we show the relative performance for computing a batch of seven FFTs.[2] In this scenario, benchFFT will repeat the calculation about 14,000 times to reach the requested 100 ms runtime. Here, our 4096-point FFT is within 5% of IPP's performance, and, over all 58 sizes, we show a 20% speed advantage.

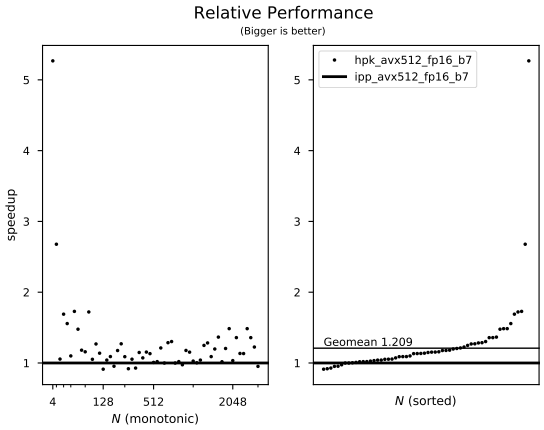

Figure 7: Performance for AVX512 half precision for a batch of seven transforms.

5.2 Single Precision

Moving on to single precision, we show in Figure 8 that our performance relative to MKL for AVX2 is 1.6X.

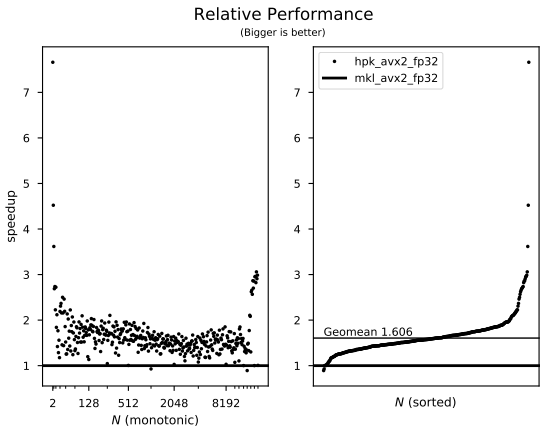

Figure 8: Performance for AVX2 single precision

In Figure 9, we show performance results for AVX512, with speedup over 1.3X.

[2]Yes, we knew this would be Figure 7 when we chose the batch size. Our only regret is not adding five more footnotes before this one.

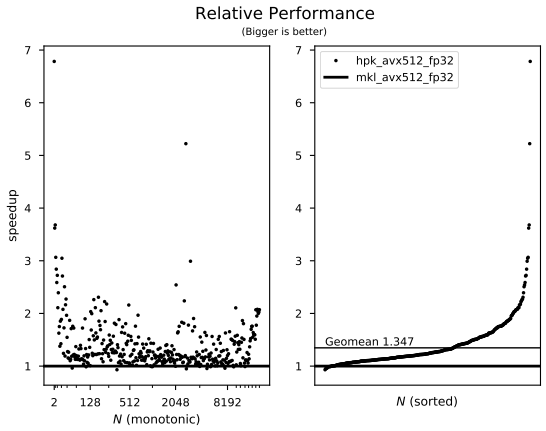

Figure 9: Performance for AVX512 single precision

We can also compare our AVX512 performance against our AVX2, and we observe that a few lengths are actually slower with AVX512. For example, the worst case is for a length of 44, which has a performance ratio of 0.67 vs. AVX2. Four complex points are 32B, the size of an AVX2 register, so it is wasteful to use a 64B AVX512 register to hold four points. Overall, the AVX512 speedup is 1.20 and would be higher if we were to eliminate these outliers.

We note that vector length extensions, AVX512VL, are supported by hardware and allow us to use 32B SIMD registers while also taking advantage of the instruction set benefits of AVX512. We do exactly this for small, but possibly more important, sizes such as 48, for which AVX512VL gives us a 3% speedup over AVX2. But we have not studied this with any rigor and not at all for length 44, with the belief that very few use this size. It's not used because, having a factor of 11, it's slower than nearby lengths, and it's slower still because we think nobody uses it.

5.3 Double Precision

The AVX2 double precision results are shown in Figure 10. In the geometric mean, we show a speedup of over 1.4X.

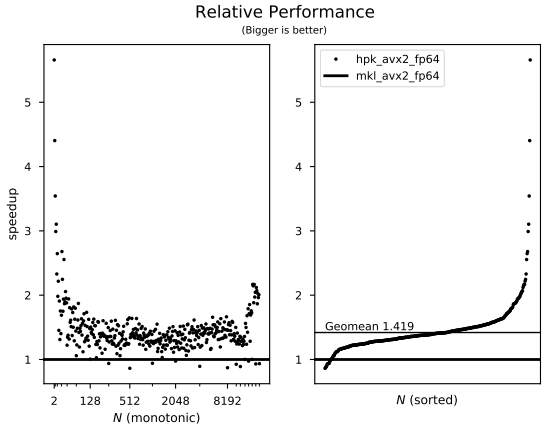

Figure 10: Performance for AVX2 double precision

Finally, we show AVX512 double precision performance results in Figure 11, with geomean speedup over 1.3X. We note that for lengths 29400 and 38400, which we highlighted earlier in the accuracy discussion, our performance relative to MKL is 1.6X and 1.2X, respectively. Thus, the better accuracy we achieved did not come at the cost of reduced performance.

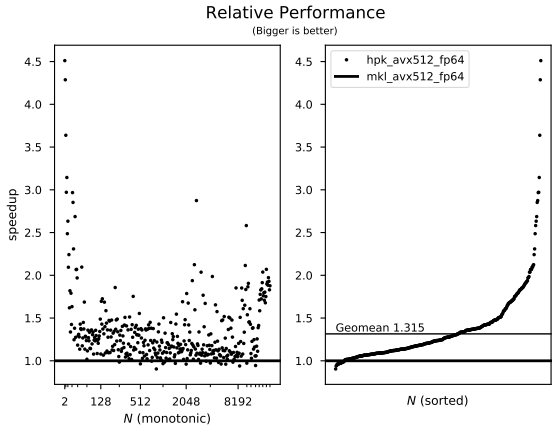

Figure 11: Performance for AVX512 double precision

We can again compare SIMD instruction sets using our library. Overall, the speedup of AVX512 over AVX2 is 1.26. In some cases our AVX512 performance is lower, and again this arises when we use 64B registers that are only half full. Two complex points are 32B, and, indeed, the biggest slowdown is for a transform of length 22, having performance ratio 0.81. Again, this is easily improved using AVX512VL, and so observations and studies as to which FFT lengths are important to high performance computing applications would be most welcome.

6 FUTURE WORK

Our first priority is to enable multithreading with OpenMP. Though many developers parallelize at the application level and hence prefer a single-threaded FFT kernel, others will appreciate an FFT compute object that automatically and effectively uses multiple threads. Our sense is that this is most valuable to those computing three-dimensional transforms, and so we will begin there.

We also intend to add support to `hpk::fft` for other hardware architectures. AArch64 with Scalable Vector Extensions would be of practical value and would afford us the opportunity to study this instruction set in depth. It would be interesting to compare and contrast the functionality it provides with that of AVX512. Though this work is no small task, we do believe our software architecture, which already supports AVX2, AVX512, and AVX512_FP16, will not require large-scale refactoring to support another target ISA.

Of course, various hardware accelerators are of interest to us. GPUs are an obvious choice, and so are FPGAs. We would also be interested in exploring numerical formats beyond the IEEE half, single, and double precisions we currently support. Special purpose hardware continues to evolve and innovate, and there are no doubt opportunities out there that we have yet to discover.

7 DOWNLOADING

High Performance Kernels for FFT, along with documentation, is available at

`https://hpkfft.com`

ACKNOWLEDGMENTS

The authors dedicate this paper to the heroes of Ukraine and would like to thank everyone who has assisted them in their time of need.

REFERENCES

[1] Måns I. Andersson, N. Arul Murugan, Artur Podobas, and Stefano Markidis. 2022. Breaking Down the Parallel Performance of GROMACS, a High-Performance Molecular Dynamics Software. https://doi.org/10.48550/ARXIV.2208.13658

[2] Brian Austin, Chris Daley, Douglas Doerfler, Jack Deslippe, Brandon Cook, Brian Friesen, Thorsten Kurth, Charlene Yang, and Nicholas J. Wright. 2018. A Metric for Evaluating Supercomputer Performance in the Era of Extreme Heterogeneity. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE Computer Society, Los Alamitos, CA, 63–71. https://doi.org/10.1109/PMBS.2018.8641549

[3] Alan Ayala, Stanimire Tomov, Azzam Haidar, and Jack Dongarra. 2020. heFFTe: Highly Efficient FFT for Exascale. In *Computational Science – ICCS 2020*. Springer International Publishing, Amsterdam, Netherlands, 262–275. https://doi.org/10.1007/978-3-030-50371-0_19

[4] Leo Bluestein. 1970. A linear filtering approach to the computation of discrete Fourier transform. *IEEE Transactions on Audio and Electroacoustics* 18, 4 (1970), 451–455. https://doi.org/10.1109/TAU.1970.1162132

[5] E. Oran Brigham. 1974. *The Fast Fourier Transform.* Prentice-Hall, Englewood Cliffs, NJ, USA.

[6] Marshall Cline, Bjarne Stroustrup, et al. 2023. C++ FAQ. https://isocpp.org/wiki/faq/friends#member-vs-friend-fns

[7] James W. Cooley. 1987. How the FFT gained acceptance. *IEEE Signal Processing Magazine* 9 (1987), 10–13.

[8] James W. Cooley and John W. Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* 19, 90 (1965), 297–301.

[9] Matteo Frigo and Steven G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231. https://doi.org/10.1109/JPROC.2004.840301

[10] Matteo Frigo and Steven G. Johnson. 2023. *benchFFT*. MIT. https://www.fftw.org/benchfft

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, Boston, MA, USA.

[12] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. 1984. Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine* 1, 4 (1984), 14–21. https://doi.org/10.1109/MASSP.1984.1162257

[13] Intel 2022. *Intel Integrated Performance Primitives Developer Reference.* Intel. https://www.intel.com/content/www/us/en/develop/documentation/ipp-dev-reference/top.html

[14] Intel 2022. *Intel oneAPI Math Kernel Library Data Parallel C++ Developer Reference.* Intel. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-mkl-dpcpp-developer-reference/top.html

[15] Intel 2023. *Intel Intrinsics Guide.* Intel. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[16] Matthew D. Jones, Joseph P. White, Martins Innus, Robert L. DeLeon, Nikolay Simakov, Jeffrey T. Palmer, Steven M. Gallo, Thomas R. Furlani, Michael T. Showerman, Robert Brunner, Andry Kot, Gregory H. Bauer, Brett M. Bode, Jeremy Enos, and William T. Kramer. 2017. Workload Analysis of Blue Waters. *CoRR* abs/1703.00924 (2017), 1–106. http://arxiv.org/abs/1703.00924

[17] Philip LaZebnik and Joe Menosky. 1991. Star Trek: The Next Generation, S5.

[18] Scott Meyers. 2014. *Effective Modern C++.* O'Reilly Media, Sebastopol, CA, USA.

[19] Alexander Stepanov and Paul McJones. 2009. *Elements of Programming.* Addison-Wesley Professional, Boston, MA, USA. http://elementsofprogramming.com

[20] Bjarne Stroustrup and Herb Sutter. 2023. C++ Core Guidelines. http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[21] Daniel Towner. 2022. *Intel AVX512-FP16 Technology Guide.* Technical Report. Intel. https://networkbuilders.intel.com/solutionslibrary/intel-avx-512-fp16-instruction-set-for-intel-xeon-processor-based-products-technology-guide

[22] Andrew Turner and Alastair Basden. 2022. HPC-JEEP: Energy Usage on ARCHER2 and the DiRAC COSMA HPC services. https://doi.org/10.5281/zenodo.7137390

Table 1: FFT lengths for half precision experimental results

4	24	60	108	180	256	360	512	648	900	1080	1440	1920	2400	3240
8	32	64	120	192	288	384	540	720	960	1152	1500	1944	2700	4096
12	36	72	128	216	300	432	576	768	972	1200	1536	2048	2916	
16	48	96	144	240	324	480	600	864	1024	1296	1620	2160	3000	

Table 2: FFT lengths for single and double precision results

2	32	78	144	234	360	528	800	1210	1764	2560	3584	5184	7000	12000	98304
3	33	80	150	240	375	540	840	1225	1792	2592	3600	5292	7056	14000	114688
4	35	81	154	243	384	550	864	1232	1800	2600	3780	5376	7168	15000	122880
5	36	84	156	245	385	560	875	1260	1815	2640	3840	5400	7200	16000	131072
6	39	88	160	250	390	576	880	1280	1872	2688	3888	5488	7500	16384	147456
7	40	90	162	252	392	588	896	1296	1920	2700	3920	5500	7680	17500	163840
8	42	91	165	256	396	600	900	1300	1944	2704	4000	5600	7776	18000	180224
9	44	96	168	260	400	624	924	1320	1960	2744	4032	5625	7840	20000	196608
10	45	98	175	264	405	630	936	1344	1980	2800	4096	5760	7920	21000	229376
11	48	99	176	270	420	640	945	1352	2000	2880	4116	5832	8000	24000	245760
12	49	100	180	280	432	648	960	1372	2016	2916	4200	5880	8064	29400	262144
13	50	104	182	288	440	660	968	1400	2048	2970	4320	6000	8100	32000	294912
14	52	105	189	294	441	672	972	1440	2080	3000	4400	6048	8192	32768	327680
15	54	108	192	297	448	675	990	1500	2100	3024	4480	6144	8400	36864	360448
16	55	110	195	300	450	700	1000	1512	2160	3072	4500	6272	8640	38400	393216
18	56	112	196	308	462	720	1008	1536	2200	3120	4536	6300	8748	40960	458752
20	60	117	198	312	468	728	1024	1540	2205	3136	4608	6400	9000	45056	491520
21	63	120	200	315	480	729	1040	1560	2240	3200	4704	6480	9072	49152	524288
22	64	125	208	320	490	735	1050	1568	2250	3240	4725	6561	9216	53248	589824
24	65	126	210	324	495	756	1080	1584	2268	3300	4800	6600	9240	57344	655360
25	66	128	216	330	500	768	1100	1600	2304	3360	4860	6720	9360	61440	720896
26	70	130	220	336	504	770	1120	1620	2352	3375	4900	6750	9600	65536	786432
27	72	132	224	343	512	780	1152	1680	2400	3456	5000	6804	9720	73728	917504
28	75	135	225	350	520	784	1176	1694	2500	3500	5040	6860	9800	81920	983040
30	77	140	231	352	525	792	1200	1728	2520	3528	5120	6912	10000	90112	1048576