Electronic preprint; submitted to Parallel Computing (Elsevier)

Intel Cilk Plus for Complex Parallel Algorithms: "Enormous Fast Fourier Transforms" (EFFT) Library

Ryo Asai and Andrey Vladimirov
Colfax International

February 5, 2015

Abstract

In this paper we demonstrate the methodology for parallelizing the computation of large one-dimensional discrete fast Fourier transforms (DFFTs) on multi-core Intel Xeon processors. DFFTs based on the recursive Cooley-Tukey method have to control cache utilization, memory bandwidth and vector hardware usage, and at the same time scale across multiple threads or compute nodes. Our method builds on single-threaded Intel Math Kernel Library (MKL) implementation of DFFT, and uses the Intel Cilk Plus framework for thread parallelism. We demonstrate the ability of Intel Cilk Plus to handle parallel recursion with nested loop-centric parallelism without tuning the code to the number of cores or cache metrics. The result of our work is a library called EFFT that performs 1D DFTs of size 2^N for $N \geq 21$ faster than the corresponding Intel MKL parallel DFT implementation by up to 1.5x, and faster than FFTW by up to 2.5x. The code of EFFT is available for free download under the GPLv3 license.

This work provides a new efficient DFFT implementation, and at the same time demonstrates an educational example of how computer science problems with complex parallel patterns can be optimized for high performance using the Intel Cilk Plus framework.

Table of Contents

1. Introduction

The discrete Fourier transform (DFT) is a universal tool in science and technology. From image processing [2] to finding distant astronomical objects [3], uses for the Fourier transform are countless and diverse. Research efforts into efficient numerical Fourier transform algorithms, dubbed discrete fast Fourier transforms (DFFTs), are numerous (e.g., [4]) and have led to great improvements in the performance of Fourier transforms in a wide range of configurations on many architectures (e.g., [5, 6]).

The definition of one-dimensional discrete Fourier transform (1D DFT) is given by the expression

$$F_k = \sum_{n=0}^{N-1} x_n \cdot \exp\left[-i\frac{2\pi kn}{N}\right], \tag{1}$$

where F_k is the complex k-th coefficient in the transform, x_n is the n-th data point, and N is the length of the transformed sequence.

If we were to construct a computer algorithm for DFT calculation using Equation (1) directly, the number of arithmetic operations (and memory accesses) would scale as $O(N^2)$. This quickly becomes ineffective even for small N, and for our domain of interest, $N \gtrsim 10^9$ (see below), direct application of Equation (1) is completely useless. In order to improve the performance, practical implementations of DFTs use optimizations that weaken the complexity scaling and qualify the algorithm as DFFT (discrete *fast* Fourier transform).

Most practical applications require either one-dimensional (1D), or multi-dimensional DFFTs with up to a few thousand data values in each dimension. In this paper, we focus on the relatively exotic domain of 1D DFTs of data sets with a few billion data values (i.e., $N \gtrsim 10^9$). This domain of DFTs has application in astronomy (e.g., [3]), however, because of its rare occurrence in other fields, existing tools for performing such DFTs are not tuned for multi-core CPUs and perform sub-optimally.

In this paper we describe a new efficient tool for such large 1D DFFTs in shared memory. We call this tool the EFFT library (the acronym stands for "Enormous Fast Fourier Transforms"). This library is optimized for use on multi-core Intel processors. EFFT uses the parallel framework Intel Cilk Plus in combination with the single-threaded implementation of small DFFTs from the Intel Math Kernel Library (MKL) [7]. EFFT performs significantly faster than the two leading DFFT libraries that we tested: FFTW [8] and MKL with little increase in memory footprint and with no loss in accuracy. However, unlike those libraries, EFFT supports only 1D transforms of size $N = M \times 2^s$, and is optimized only for very large values of N.

In addition to the practical importance of the EFFT library, its development process has great educational value. That is because the parallel DFFT algorithm is an excellent example of a problem with complex pattern of parallelism. As such, it is an ideal problem for demonstrating the strength of Intel Cilk Plus framework in effectively parallelizing difficult problems with little need for programmer input. Furthermore, our implementation of EFFT showcases multiple techniques for optimization for Intel Xeon processors, which are effective for not only DFFT, but a diverse set of algorithms.

In Section 2 we discuss the sub-optimal performance of large multi-threaded 1D DFTs in MKL and the possible route to resolution of this inefficiency through the use of the Cooley-Tukey recursive algorithm in combination with parallel framework Intel Cilk Plus. In Section 3 we discuss the implementation of EFFT and the optimization consideration that allow us to achieve better performance than MKL. Section 4 contains brief instructions on using the EFFT library, and Section 5 reports the tuning methodology and performance benchmarks for a range of array sizes on a multi-core processor. Discussion in Section 6 touches on the utility of Intel Cilk Plus for complex parallelism and on the applicability of our approach to the Intel Xeon Phi coprocessor architecture.

The result of our work, the EFFT library, is available for free download at the Colfax Research web site [1] under the GPLv3 license.

2. Parallelizing the 1D DFFT Calculation

2.1. Prior Art: Intel MKL DFT

Intel Math Kernel Library (MKL) is a highly optimized, industry-leading mathematics function library. MKL supports DFTs in serial, multi-threaded and cluster implementations. As we are mostly interested in multi-threaded 1D large transforms, in this section we benchmark the MKL performance for this problem domain.

Listing (1) shows a basic example code that carries out a single-threaded, one-dimensional, real data DFFT in single precision on the data array of the size $N = 2^{28}$.

```
// (1) creating the descriptor for single-
//     precision DFFT on real data
const int N = 1<<28;
float* data = (float*)
             _mm_malloc(N*sizeof(float), 64);
MKL_LONG fftsize = N;
DFTI_DESCRIPTOR_HANDLE* fftHandle =
                  new DFTI_DESCRIPTOR_HANDLE;
DftiCreateDescriptor(fftHandle, DFTI_SINGLE,
                     DFTI_REAL, 1, fftsize);

// (2) configuring the descriptor:
//     single-threaded transform,
//     packed permuted data format
DftiSetValue(*fftHandle,
             DFTI_NUMBER_OF_USER_THREADS, 1);
DftiSetValue(*fftHandle, DFTI_PACKED_FORMAT,
                         DFTI_PERM_FORMAT);
DftiCommitDescriptor (*fftHandle);

// (3) carrying out in-place DFT
DftiComputeForward(*fftHandle, data)
```

Listing 1: Basic MKL DFT code.

The flow of using the DFT functionality in MKL is as follows:

1. Create a `DFTI_DESCRIPTOR_HANDLE`. This object stores the parameters of the transform: precision, type, dimensions and size.
2. Set various options for the handle including the data format (layout) and the number of threads. After the change are made, the handle must be finalized with the function `DftiCommitDescriptor()` before it can be used. Once it has been committed, the handle can be used as many times as the user needs.
3. Carry out the transform using function `DftiComputeForward(...)`. This runs an in-place forward Fourier transform of the array `data`.

The format `DFTI_PERM_FORMAT` that we used is an output format that is specific to real-data transforms. With this format, the input data array contains N data points in the following order:

$$\{x_0,\ x_1,\ x_2,\ \ldots,\ x_{N-1}\}, \tag{2}$$

and the output array contains $N/2 + 1$ complex coefficients $F_k \equiv R_k + iI_k$ in the following order:

$$\{R_0, R_{\frac{N}{2}}, R_1, I_1, R_2, I_2, \ldots, R_{\frac{N}{2}-1}, I_{\frac{N}{2}-1}\}. \tag{3}$$

For $k \geq N/2$, the values of F_k can be inferred using the symmetry property expressed below by Equation (8).

Assuming that the user application must process multiple DFTs, there are two approaches to parallelizing the application:

1. Implementing multiple single-threaded Intel MKL DFTs. In this approach, each thread has its own DFTI_DESCRIPTOR_HANDLE. The parallelization is done by calling multiple instances of single-threaded `DftiComputeForward()` from multiple user threads.
2. Taking advantage of the internal threading functionality of the Intel MKL DFT. In this approach, only one DFTI_DESCRIPTOR_HANDLE is created, and the value of parameter `DFTI_NUMBER_OF_USER_THREADS` is set to the desired number of threads. In this case, the parallelization is done internally by MKL, and multiple cores work in parallel on a single transform.

The first approach is applicable when (a) all DFTs needed by the application are independent from each other, and therefore can be computed concurrently, and (b) the problem size is small enough so that T data arrays can fit in memory, where T is the number of threads used by the application. The second approach has the advantage of using far less memory, because only one data array must be stored in memory at any given time. This approach also allows to compute each DFT as fast as possible in terms of wall clock time, which is useful when the result of one DFT determines the subsequent operations in the parallel application.

Figure (1) shows the performance of the two approaches as a function of the number of threads for DFTs of size $N = 2^{28}$. The red solid line with square markers reports the performance of multiple single-threaded DFTs, and the dotted blue line with circular markers shows the performance of a single multi-threaded DFT. In both cases, the total number of threads is set using the `OMP_NUM_THREADS` environment variable. Furthermore in the former case, the number of threads for individual DFT is set to one using `mkl_set_num_threads()` as well as by setting `DFTI_NUMBER_OF_USER_THREADS` in the descriptor handle.

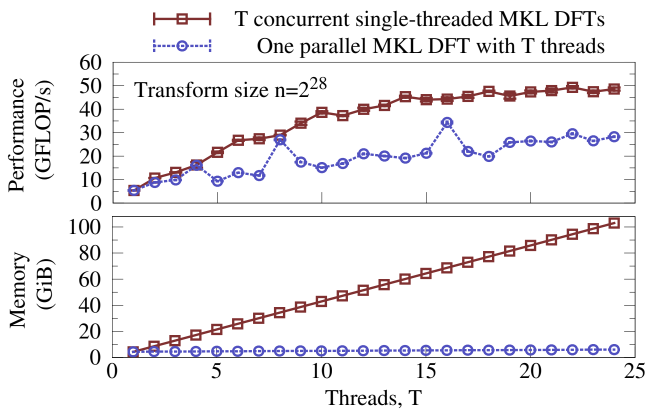

Figure 1: *Top*: Performance of two parallelization approaches as the function of the number of threads. *Bottom*: memory usage in the two approaches.

Based on performance alone, multiple single-threaded DFTs approach has a clear advantage in scalability over the internal threading approach. However, running multiple single-threaded DFTs is entirely impractical in the domain of large transforms due to its enormous memory requirement. The bottom panel of Figure (1) shows the memory usage as a function of number of threads for carrying out Intel MKL DFT. With multiple concurrent single-threaded DFTs running, the required memory increases linearly with the number of threads T. Thus, in processors with high core count and ($N \gtrsim 10^9$), the memory requirement for the multiple single-threaded DFTs approach may exceed the practical amount of usable RAM.

In this work, we develop the EFFT library for executing multi-threaded DFTs without the performance loss experienced by MKL for large numbers of threads. EFFT builds upon the single-threaded MKL functionality in combination with the Cooley-Tukey algorithm. This algorithm, described in the next section, allows to construct one large DFT from results of many smaller DFTs.

2.2. Radix-2 Cooley Tukey Algorithm

The CT algorithm originally presented in [9] is the archetype of most DFFT algorithms and is widely used and studied in order to improve the performance of Fourier transforms. In EFFT, we only use the Radix-2 version, which is a recursive algorithm that breaks down a transform of size N into two smaller transforms of size $N/2$ at each step. Note that this algorithm requires that the size $N = M \times 2^s$, where s is the number of times we intend to apply the recursive splitting.

The core idea of the Radix-2 CT algorithm is to rewrite Equation (1) as a sum of the elements at even values of n and odd values of n:

$$\begin{aligned} F_k &= \sum_{n=0}^{N/2-1} x_{2n}\cdot \exp\left[-i\frac{2\pi k(2n)}{N}\right] + \\ &\quad + \sum_{n=0}^{N/2-1} x_{2n+1}\cdot \exp\left[-i\frac{2\pi k(2n+1)}{N}\right] = \\ &= \sum_{n=0}^{N/2-1} x_{2n}\cdot \exp\left[-i\frac{2\pi kn}{N/2}\right] + \\ &\quad + e^{-i2\pi k/N}\sum_{n=0}^{N/2-1} x_{2n+1}\cdot \exp\left[-i\frac{2\pi kn}{N/2}\right] \equiv \\ &\equiv E_k + e^{-i2\pi k/N} O_k. \end{aligned} \tag{4}$$

By comparing Equation (4) with (1), one can see that the first sum is just the Fourier transform of the even-numbered elements and the second sum is the transform of the odd-numbered elements, multiplied by

$e^{-i2\pi k/N}$. The multiplication factor $e^{-i2\pi k/N}$ is often referred to as the "twiddle factor". Hereafter, we denote the even part of the transform as E_k and the odd transform as O_k.

Values of E_k and O_k need to be found only for $0 < k < N/2$, because from the periodicity of the complex exponential functions follows the symmetry

$$F_k = E_k + e^{-i2\pi k/N} O_k, \quad (5)$$

$$F_{k+N/2} = E_k - e^{-i2\pi k/N} O_k. \quad (6)$$

The CT algorithm repeats this decomposition recursively to produce smaller and smaller DFTs. The number of arithmetic operations (FLOPs) required to compute a transform of size N using the CT recursion is

$$\text{FLOPs} = \frac{5}{2} \times N \log_2 N \quad (7)$$

This definition of FLOPs conflates the low-latency operations of floating-point addition and multiplication with the long-latency transcendental arithmetic operations, and therefore cannot be easily related to the hardware performance metrics. However, for simplicity, we use this scaling to express the performance of the implementation in terms of GFLOP/s.

For DFTs of real data (i.e., $x_n = x_n^*$), the transform coefficients also have the following symmetry;

$$F_k = F^*_{(N-k-1)}. \quad (8)$$

It is trivial to demonstrate that for all DFTs, F_0 and $F_{N/2}$ are both real. This means that only coefficients F_1 to $F_{N/2-1}$ need to be stored in their full complex form, and the coefficients F_0 and $F_{N/2}$ can be stored as real values. This property allows the result of a real data transform, F_k, to be stored in the same sized memory array as the input data x_n.

2.3. Parallel Algorithm of EFFT

For our purposes, the value of the CT algorithm not only in the reduction of arithmetic complexity from $O(N^2)$ to $O(N \log N)$, but also in its ability to decompose a large Fourier transform into a number of smaller Fourier transforms. By decomposing a large transform into many smaller independent transforms, we can execute each of these transforms serially (i.e., with a single processor core) using an existing highly optimized DFT implementation. Parallelism can be achieved by distributing the small serial DFTs across multiple processor cores.

However, for the purposes of optimization, parallelism also has to be achieved in the procedures that precondition the data layout for parallel DFTs. Parallelism is also necessary in the application of Equations 5 and 6. This makes the parallelization much more challenging.

In the EFFT library we express the parallel DFFT as three stages:

I. "Scattering" of x_n into multiple contiguous arrays ("bins"), which are thereafter transformed,

II. "Processing" (i.e., performing a serial DFT on) each of the small "bins", and

III. "Reassembly", i.e., the application of Equations 5 and 6 to the "bins" in order to produce the transform coefficients F_k.

This workflow is depicted in Figure 2.

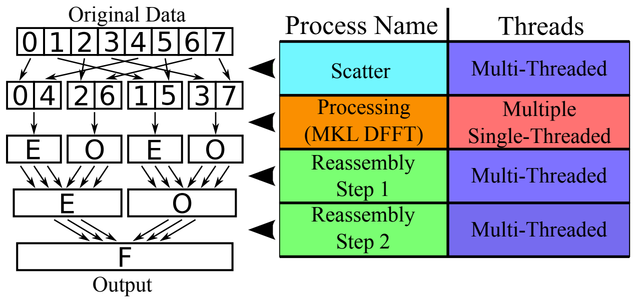

Figure 2: Workflow of EFFT with $s = 2$ splits and $b = 2^s = 4$ bins. One "scatter" stage is performed with a multi-threaded loop. It is followed by b single-threaded "processing" calls to MKL DFT function. Each pair of consecutive processed bins is "reassembled" by a multi-threaded loop. Multiple "processing" and "reassembly" calls may be running concurrently.

Parallelizing the scattering phase across multiple CPU cores is discussed in Section 3.2. As for the processing stage, data locality considerations call for a recursive algorithm, as discussed in Section 3.3. Parallelizing the processing is done by distributing the single-threaded DFTs in each bin across the CPU cores. The difficulty with parallelizing the processing stage is that the number of bins in the simplest algorithm is always a power of 2, however, the number of CPU cores is not necessarily a power of 2.

Furthermore, data locality considerations suggest that reassembly must be performed as soon as possible after processing. Also, because towards the end of the algorithms, the number of reassembled arrays decreases and eventually gets as low as 2, scalability considerations require that the reassembly operation itself contain thread parallelism inside.

It can be very difficult in a low-level parallel framework to overlap the multiple independent serial processing tasks with multiple interleaved multi-threaded reassembly tasks, especially if the number of CPU cores is not proportional to the number of tasks. We have tried, but could not come up with a satisfactory implementation using OpenMP, a framework in which the programmer is responsible for assigning specific work items to hardware threads. At the same time, we were able to express the same algorithm in Intel Cilk Plus in a short amount of time, and achieve better results than we were able to obtain with OpenMP. Section 2.4 overviews the Intel Cilk Plus framework and explains our implementation.

2.4. Intel Cilk Plus

Intel Cilk Plus [10, 11] is a powerful parallelization and vectorization framework that can effectively parallelize complex problems with very little work required on the part of the programmer. In this framework, the programmer specifies the components of the application that can be run in parallel, and the runtime library takes care of assigning computing resources (cores) to the parallel tasks. This is done via an internal scheduling mechanism based on “work stealing” to distribute parallel work-items among “workers”. Workers are a concept in Intel Cilk Plus similar to threads in other frameworks (for example, OpenMP). Because of the simplicity of the API and high degree of behind-the-scenes automation, Intel Cilk Plus can dramatically reduce the development workload and time while providing great performance.

There are only 3 keywords in Intel Cilk Plus: `cilk_for`, `cilk_spawn` and `cilk_sync`. The keyword `cilk_for` is a replacement for the C++ `for` with a hint that parallel execution is possible in this loop. `cilk_spawn` is used to launch a task represented by a function in parallel with the current program. Finally, `cilk_sync` is a barrier for all tasks spawned from the current task. Complex patterns of parallelism, including nested parallel programs and parallel regions requiring different degrees of parallelism, are fully supported.

```
cilk_for (i = 0; i < n; i++) {
  // Iterations of this loop
  // are distributed between workers
}

// Launch MyFunction
cilk_spawn MyFunction(a, b, c);
// This will run without waiting
// for MyFunction() completion
MyOtherFunction(b, a, c);

// Wait for completion of MyFunction
cilk_sync;
```

Listing 2: Summary of Intel Cilk Plus framework keywords.

However, note that this ease of use of Cilk functionality does not release the programmer from following the necessary precautions that apply to all parallelized loops. Avoiding issues such as race conditions is still the programmer’s responsibility. Intel Cilk Plus provides C++ templates referred to as reducers in order to eliminate race conditions in parallel programs with certain patterns (see, e.g., [12]).

The functionality of Intel Cilk Plus perfectly fits with the parallel pattern of EFFT. Namely, `cilk_for` can be used to parallelize the scatter phase and to effect parallelism inside of each reassembly task. `cilk_spawn` and `cilk_sync` can be used to achieve parallel recursion.

We have limited our discussion of Intel Cilk Plus functionality to those that are relevant to our later discussions. For complete documentation, visit the Intel C++ Composer Reference Manual [13] and the Intel Cilk Plus Web resources [10, 11].

3. Optimizing EFFT

This section focuses on the steps we took to parallelize and optimize the EFFT library.

Before we examine the code, we will first describe some of the terminology used in this text and in the code. The parameter we call "number of splits", represented in codes by variable `numofsplits`, and in the text by the variable s, is the number of Radix-2 CT algorithm steps applied to the transform. For example, with 3 splits, we apply the Radix-2 CT algorithm 3 times, thus decomposing the transform of size N into $b = 2^3 = 8$ smaller transforms of size $N/8$. These smaller transforms are stored in segments of a larger "scratch array". We refer to these segments as "bins", which in the code is represented by variable `numofbins` and in the text by b. The number of bins is related to the number of splits as $b = 2^s$ or $s = \log_2 b$.

3.1. Initialization

EFFT is a C++ library that implements a class `EFFT_Transform`. The constructor of this class takes care of data allocation and initialization for the transform. Majority of the initialization code is self-explanatory, but we wanted to highlight two important details related to performance optimization.

The first detail is data allocation. The initialization function in EFFT automatically allocates both the data storage array as well as the necessary temporary storage array. The storage array exists as long as the encompassing EFFT class exists. We retain this array to reduce the overhead of the `Transform()` method. Additionally, by allocating the memory internally, we ascertain that (i) the data container is aligned on a 64-byte boundary, and (ii) that the first touch to the data container is performed in a parallel region by the MKL DFT initialization function. Both of these details are important for subsequent performance of the FFT.

The second important aspect of EFFT initialization is MKL DFT handle creation. Since the time required to create a handle is often longer than the actual transform itself, it is important to create the handle once and retain it for all subsequent transforms. Furthermore, in order to implement the most efficient MKL usage mode, each Intel Cilk Plus worker creates its own handle to be used in all subsequent transforms.

Listing (3) shows the portions of the initialization that we have discussed.

```
// Data allocation on 64 byte boundary
dataPtr =
   (float*)_mm_malloc(size*sizeof(float), 64);
scratchPtr =
  (float*) _mm_malloc(size*sizeof(float), 64);

// Handle creation
DFTI_DESCRIPTOR_HANDLE* fftHandle =
 (DFTI_DESCRIPTOR_HANDLE*)malloc(numthreads*
                 sizeof(DFTI_DESCRIPTOR_HANDLE));

// Spawning a parallel region
cilk_for (int i = 0; i < numworkers; i++){
  int wid = __cilkrts_get_worker_number();
  mkl_set_num_threads(1);
  // Preparing small MKL FFT
  MKL_LONG fftsize = binsize;
  // One handle per worker
  DftiCreateDescriptor(&fftHandle[wid],
          DFTI_SINGLE, DFTI_REAL, 1, fftsize);
  DftiSetValue(fftHandle[wid],
              DFTI_NUMBER_OF_USER_THREADS, 1);
  DftiSetValue(fftHandle[wid],
        DFTI_PACKED_FORMAT, DFTI_PERM_FORMAT);
  DftiCommitDescriptor (fftHandle[wid]);
}
```

Listing 3: Data allocation and handle creation a initialization.

3.2. Scattering Stage

The purpose of the scatter phase of the transform is to reorganize the data set into a temporary data array, where the elements for small DFTs are listed contiguously. In terms of the CT algorithm, this is the phase where we separate the even-numbered elements and odd-numbered elements.

Listing (4) shows a basic, unoptimized scatter that is equivalent to one step in the Radix-2 CT algorithm.

```
for (int i = 0; i < N; i+=2) {
  temparray[i] = data[2*i];
  temparray[i+N/2] = data[2*i+1];
}
```

Listing 4: Basic recursive scatter.

This basic version of scatter can be applied recursively to arrive at the reorganized array we require, but has extremely poor performance due to high number of memory accesses. In order to carry out the Radix-2 CT algorithm s times, or to do s splits, this recursive scatter will have to read $2 \cdot s \cdot N$ memory addresses. Thus, in order to gain performance it would be ideal to do the full scatter in a single pass.

In our EFFT code, we use the fact that s is generally small and precompute the pattern of mapping from the original data array to the scratch array. This mapping is based on the bit reversal principle[9]. Listing (5) shows a (still unoptimized) variant of our that version. In that code, we also use $cilk\_for$ to parallelize the scattering process.

```
int scatter_index[numofbins];
find_scatter_index(scatter_index,numofsplits);
cilk_for(int j = 0; j < numofbins; j++)
  for (long i = 0; i < binsize; i++)
    tempArray[scatter_index[j]*binsize+i] =
                                 dataArray[i*numofbins+j];
```

Listing 5: Unoptimized scatter.

There are several techniques we can employ to optimize this scatter code for parallelism and improve its performance. First technique is to permute the j and the i-loops. This is helpful because in a target parameter domain of EFFT ($N \gtrsim 10^9$), practical values of $b \equiv$ `numofbins` are much smaller than $N/s \equiv$ `binsize`. Therefore, in modern multi-core processors, parallelizing the j-loop with b iterations can potentially lead to the problem of not having enough parallelism. For instance, if we had $T = 24$ workers and $b = 32$ bins, the runtime system would first distribute 24 bins across the 24 workers. Once this work is done, only $32 - 24 = 8$ of the 24 workers will be utilized for the remainder of the calculation. Therefore, permuting the loops to bring the i loop outside and parallelze it ensures that there is enough parallelism in the code.

However, the loop permutation has the negative effect of losing unit-stride access in i in the inner loop. It is true that we gain unit-stride in j in exchange for this loss, but for this particular implementation of the scatter, unit-stride in i is of greater importance. This is due to the fact that the variable i is multiplied by $b \equiv$ `numofbins`, while j (or, rather, `scatter_index[j]`) is multiplied by `binsize` $\equiv N/b$. Again, the b is normally much smaller than N/s, so memory accesses with consecutive values of i are located closer to each other than accesses consecutive in j. Thus having i in the inner loop is more optimal.

In order to get parallelism in i while keeping a unit-stride access in i, we implement the technique called loop tiling. This technique involves strip-mining the inner loop and permuting the outer two loops. The strip-mining operation expresses a single loop as two nested loops, of which the outer loop iterates with a stride referred to as "Tile", and the inner loop iterates incrementally within the "Tile". Listing (6) shows our final optimized code. The value `iTile=16` is obtained empirically and provides the best performance in most cases.

```
int scatter_index[numofbins];
find_scatter_index(scatter_index, numofsplits);
const long iTILE=16;
long ii;
cilk_for (ii = 0; ii < binsize; ii += iTILE)
  for(long j = 0; j < numofbins; j++)
    for (long i = ii; i < ii+iTILE; i++)
      tempArray[scatter_index[j]*binsize+i] =
                              dataArray[i*numofbins+j];
```

Listing 6: Optimized scatter code with tiling.

In order to understand why Listing 6 is optimal, consider the locality of data accesses. Each worker operates on a different value of `ii` (in practice, each worker will get a contiguous chunk of values of `ii` to process). For each value of `ii` and each value of `j`, the worker reads `iTile==16` elements from `dataArray` with a stride of `numofbins`. Then these elements are written contiguously into `tempArray`. Contiguous access to `tempArray` is good, however, in the process of reading `dataArray`, the worker had touch 16 cache lines to read only 16 words. Luckily, the loop in `i` is short enough so that for the next value of j, these cache lines are re-used. Therefore, the application can achieve a significant fraction of memory bandwidth. At the same time, the outer parallel loop has enough iterations to scale across tens of threads available in modern CPUs.

3.3. Processing/Parallel Recursion

In order to parallelize the processing and reassembly phases, we have combined them into a single parallel recursion tree. This framework is shown in Listing 7.

```
void ProcessAndReassemble(
         DFTI_DESCRIPTOR_HANDLE* fftHandle,
         float* array, float* temp,
         const long n, const long binsize) {
  const long size = n/2;

  if (n > binsize) {
    cilk_spawn // Create parallel task
      ProcessAndReassemble(fftHandle,
                array, temp, size, binsize);
    ProcessAndReassemble(fftHandle,
                &array[size], &temp[size],
                             size, binsize);
    cilk_sync; // Wait for spawned task

    // CT algorithm to combine evens and odds
    // ReassemblePair() is multithreaded
    ReassemblePair(temp, size);
  } else {
    // Process the small enough array
    // with serial MKL implementation of DFT
    int wid = __cilkrts_get_worker_number();
    DftiComputeForward(fftHandle[wid], temp);
  }
}
```

Listing 7: Parallel recursive processing and reassembly in EFFT.

The processing call is initially applied to the array of size n=N. This call subsequently recurses into two instances of itself with n reduced by a factor of 2. One of the recursive calls is placed in an Intel Cilk Plus task using the keyword cilk_spawn, thus effecting parallel recursion. Recursion stops then n is reduced to the target binsize, and processing with serial MKL DFT is applied in parallel to the multiple small “bins”. After the processing of two adjacent bins is complete (as indicated by the return of the cilk_sync call), the reassembly function is called on these two bins. Reassembly has thread parallelism inside, as explained in Section 3.4

To see why we chose to incorporate reassembly in the same parallel recursion tree as processing, consider a case where the architecture allows for $T = 24$ threads, and we do $s = 5$ splits to get $b = 32$ bins. The code starts the first 24 serial DFTs in parallel, as desired. However, after the first 24 bins are processed (transformed), the rest of the DFTs can occupy only 8 threads. In order to utilize all available CPU resources, the remaining 16 threads should work on reassembly while the other 8 finish the rest of the DFTs. Thus, in order to utilize all available hardware in a parallel setting, it is beneficial to combine reassembly and processing in the same tree.

Implementing a parallel framework to execute such a parallel algorithm is not trivial. One of the complications is that the number of workers available for the reassembly varies as DFTs get finished. Keeping track of how many threads are available in a parallel region is complicated, as it requires monitoring which bins can be assembled and by which thread. Another issue is that multiple calls to multi-threaded reassembly function may run simultaneously. Controlling the order of their execution and preventing the system from oversubscription of threads can be challenging. In cases like this, the Intel Cilk Plus framework can be a convenient solution. Work items are assigned to workers automatically through the runtime library’s high performing scheduling algorithm.

3.4. Reassembly

The purpose of the reassembly phase is to apply Equations (5) and (6) to assemble the results of smaller DFTs to produce the result of the larger decomposed FFT. This stage is performed by the function ReassemblePair() called from ProcessAndReassemble() in Listing 7.

The reassembly component uses the Radix-2 CT algorithm to recombine the results of the smaller DFTs. Listing 8 shows a basic (i.e., unoptimized) implementation of reassembly that applies a singe step of Radix-2 CT algorithm to combine two bins into one.

In Listing 8, evens and odds are the FFT result of even- and odd-numbered elements, respectively (E_k and O_k), target is the target location to stored the combined result (F_k), and *size* is the size of the evens and odds arrays. Note that the first two elements are dealt with as special cases, because those are the real parts of F_0 and $F_{N/2}$ instead of the real-imaginary components of $F_k \equiv R_k + iI_k$ like the rest (see Equation 3).

```
void BasicReassemble(float* evens, float* odds,
         float* target, const long int size) {
  const float trigconst = -3.14159265359f/size;
  target[0      ]=evens[0] +odds[0];
  target[1      ]=evens[0] -odds[0];
  target[size   ]=evens[1];
  target[size+1L]=-odds[1];

  for(long int k = 1L; k < size/2L; k++) {
    float cosk = cosf(k*trigconst);
    float sink = sinf(k*trigconst);
    target[2*k]=evens[2*k]+odds[2*k]*cosk
                      -odds[2*k+1]*sink;
    target[2*k+1]=evens[2*k+1]+odds[2*k]*sink
                        +odds[2*k+1]*cosk;
    target[2*size-2*k]=evens[2*k]
             -odds[2*k]*cosk+odds[2*k+1]*sink;
    target[2*size-2*k+1]=-evens[2*k+1]
             +odds[2*k]*sink+odds[2*k+1]*cosk;
  }
}
```

Listing 8: Basic (not optimized) reassembly of two arrays into one using Equation (5).

Following equation (5), the code adds the k-th element of the `evens` to the k-th element of the `odds` multiplied by the twiddle factor.The second half of the target array, which corresponds to $k \geq N/2$ (which are not in the `evens` and `odds` arrays), is calculated by taking advantage the symmetry expressed by Equation (8).

At first glance this appears to be a problem that is easily vectorized, since the result is simply a superposition of elements of two arrays multiplied by trigonometric functions, which can be vectorized. However, because the arrays contain complex numbers with real and imaginary parts interleaved, access to the potentially vectorizable data has a stride of 2. This situation is difficult for the compiler to handle, and we have found that the following code modification improves automatic vectorization and, along with it, application performance.

First, in order to vectorize the calculation of trigonometric functions, we strip-mine the array in k. This operation expresses the loop in k as two nested loops in kk (iterating with a stride of 64) and a loop in k (iterating with a stride of 1 through 64 iterations). The second step is un-fusing the loop over the trigonometric functions from the loop over array elements (see Listing 9). The reason for strip-mining the loop is to restrict the size of the container for precomputed trigonometric functions to a small enough value that does not cause the eviction of `evens` and `odds` from caches. Note that with the strip-mined loop, we have to handle $k = 0$ as an exception (see output format given by Equation 3). This also necessitates a separate loop for processing elements from $k = 1$ to $k = kTILE - 1$, because the automatically vectorized loops in lines 18 and 24 of Listing 9 is built for complete tiles of size $kTILE$. Separate loop for the first tile introduces redundant code, however, it allows us to avoid branches in the main loops in lines 18 and 24, which would ruin the overall performance of the application.

```
 1 const long kTILE = 64L;
 2
 3 // ...omitted the code handling 0 and N/2
 4
 5 for(long k = 1; k < kTILE; kk+=kTILE) {
 6   // Elements from 1 to kTILE-1
 7   // are handled separately
 8 }
 9
10 #define ALIGNED __attribute__((aligned(64)))
11 cilk_for(long kk = kTILE;
12                 kk < size/2L; kk+=kTILE) {
13
14   // Unfused loop to precompute trigonometric
15   // functions with vector operations
16  float coslist[kTILE] ALIGNED,
17        sinlist[kTILE] ALIGNED,
18  for (int i = 0; i < kTILE; i++) {
19   coslist[i] = cosf((kk+i)*trigconst);
20   sinlist[i] = sinf((kk+i)*trigconst);
21  }
22
23  // Iterating within the tile
24  for (long k = kk; k < kk+kTILE; k++ {
25   target[2*k]=evens[2*k]+odds[2*k]*coslist[k]
26                    -odds[2*k+1]*sinlist[k];
27   target[2*k+1]=evens[2*k+1]+
28                    odds[2*k]*sinlist[k-kk]
29                 +odds[2*k+1]*coslist[k-kk];
30    //...
31  }
32 }
```

Listing 9: Improved (still not optimal) reassembly code. Strip-mining helps to vectorize the trigonometric functions.

The second optimization of vectorization relies on the Intel Cilk Plus array notation to assist the vectorization of operation on strided data. Indeed, the `for`-loop in line 24 of Listing 9 mixes data with a stride of 1 (in `sinlist` and `coslist`) and stride 2 (`evens`, `odds` and `target`). We have found that re-writing this loop with array notation as shown in Listing 10 improves the efficiency of automatic vectorization. Array notation is not a standard part of the C++ language; it is a language extension provided by the Intel Cilk Plus framework. It is supported by the Intel C++ compiler and by GCC [14].

```
cilk_for(long kk = kTILE;
                  kk < size/2L; kk+=kTILE) {

 float coslist[kTILE] ALIGNED,
       sinlist[kTILE] ALIGNED,
       evenrek[kTILE] ALIGNED,
       evenimk[kTILE] ALIGNED,
 // ...

#pragma simd
#pragma vector aligned
  for (int i = 0; i < kTILE; i++) {
    sinlist[i] = sinf(theta+i*trigconst);
    coslist[i] = cosf(theta+i*trigconst);
  }

  // Iterating within the tile (array notation)

  // Gather elements with a stride of 2
  evenrek[:] = evens[kk  :kTILE:2];
  evenimk[:] = evens[kk+1:kTILE:2];
  oddrek [:] = odds [kk  :kTILE:2];
  oddimk [:] = odds [kk+1:kTILE:2];

  // Reassembl & scatter into stride-2 array
  target[kk  :kTILE:2] = evenrek[:] +
   coslist[:]*oddrek[:]-sinlist[:]*oddimk[:];
  target[kk+1:kTILE:2] = evenimk[:] +
   sinlist[:]*oddrek[:]+coslist[:]*oddimk[:];

    //...
}
```

Listing 10: Further optimized (still not optimal) reassembly code. Array notation helps the compiler to vectorize operations with strided data accesses.

Finally, one more target of optimization in the reassembly code is memory access. The unoptimized code in Listing 10 combines two elements from input arrays and writes out the result into a separate output array. This means that in this code, reassembly is done *out-of-place*. Indeed, the array index of the destination element F_k is not the same as the index of the source elements E_k and O_k. Diagram (3) demonstrates this problem.

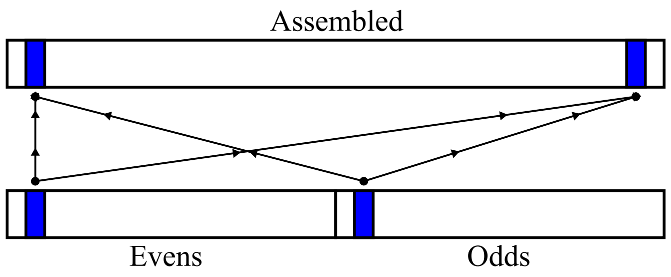

Figure 3: Diagram showing the memory locations in a CT assembly of an element. Because of the location of the elements, this algorithm can not be done in-place.

However, this difficulty can be overcome by performing reassembly in pairs of elements. Due to the symmetry of the operation, by combining two different steps in reassembly, it is possible to get an assembly step where the memory location for the input is the same as the destination memory location for the output. Indeed, upon the assembly of E_k and O_k, we have to write F_k and F_{N-k-1}. If at the same time we assemble $E_{N/2-k-1}$ and $O_{N/2-k-1}$, then we can write $F_{N/2-k-1}$ and $F_{N-(N/2-k-1)-1} = F_{N/2+k}$. The offset of the latter element coincides with the offset of O_k. Therefore, it is possible to read the values of E and O elements and write the values of the corresponding F elements into the same memory locations. Diagram (4) shows this assembly-in-pair scheme. Using this method, in the optimized version of element combination function (Listing 11) we perform reassembly *in-place*. By doing this, we reduce the number of memory accesses by a factor of 2.

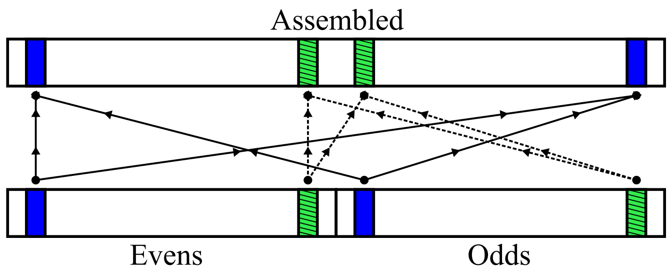

Figure 4: Diagram showing the memory locations in a CT assembly in pairs. With this method the memory locations overlap, and in-place CT Assembly is possible.

```
cilk_for (long kk = kTILE; kk < halfsize/2L;
                              kk += kTILE) {
  float coslist[kTILE] ALIGNED,
        sinlist[kTILE] ALIGNED,
        evenrek[kTILE] ALIGNED,
        evenimk[kTILE] ALIGNED,
        //...

  // Vectorized twiddle factor precomputation
  const float theta=(float)(kk/2)*trigconst;
#pragma simd
#pragma vector aligned
  for (int i = 0; i < kTILE; i++) {
    sinlist[i] = sinf(theta+i*trigconst);
    coslist[i] = cosf(theta+i*trigconst);
  }

  // Gather stride-2 data for first pair
  evenrek[:] = evens[kk  :kTILE:2];
  evenimk[:] = evens[kk+1:kTILE:2];
  oddrek [:] = odds [kk  :kTILE:2];
  oddimk [:] = odds [kk+1:kTILE:2];

  // Reassemble first pair
  // and scatter results into stride-2 array
  evens[kk  :kTILE:2] = evenrek[:] +
   coslist[:]*oddrek[:]-sinlist[:]*oddimk[:];
  evens[kk+1:kTILE:2] = evenimk[:] +
   sinlist[:]*oddrek[:]+coslist[:]*oddimk[:];

  // Start gather second pair (mirror)
  oddmirrek[:] = odds[size-kk  :kTILE:-2];
  oddmirimk[:] = odds[size-kk+1:kTILE:-2];

  // Finish reassembly of first pair
  odds[size-kk  :kTILE:-2] =  evenrek[:] -
   coslist[:]*oddrek[:]+sinlist[:]*oddimk[:];
  odds[size-kk+1:kTILE:-2] = -evenimk[:] +
   sinlist[:]*oddrek[:]+coslist[:]*oddimk[:];

  // Finish gathering second pair (mirror)
  evenmirrek[:] = evens[size-kk  :kTILE:-2];
  evenmirimk[:] = evens[size-kk+1:kTILE:-2];

  // Reassemble second pair of elements
  // and scatter results into stride-2 array
  evens[size-kk  :kTILE:-2]= evenmirrek[:] -
                  sinlist[:]*oddmirrek[:]
                 +coslist[:]*oddmirimk[:];
  evens[size-kk+1:kTILE:-2]= evenmirimk[:] -
                  coslist[:]*oddmirrek[:]
                 -sinlist[:]*oddmirimk[:];
  odds [kk       :kTILE: 2]= evenmirrek[:] +
                  sinlist[:]*oddmirrek[:]
                 -coslist[:]*oddmirimk[:];
  odds [kk+1     :kTILE: 2]=-evenmirimk[:] -
                  coslist[:]*oddmirrek[:]
                  -sinlist[:]*oddmirimk[:];
}
```

Listing 11: Optimized main loop in the reassembly function.

The code shown in Listing 11 is the final version that we adopted in EFFT. It is optimized with the following methods:

1. Strip-mining to precompute the trigonometric functions with vector operations,
2. Array notation to help the compiler vectorize loops with non-unit stride access, and
3. Performing reassembly on symmetrically located pairs of elements in each iteration in order to perform the job *in-place*, reducing the number of memory accesses.

Listing 11 does not contain the code for the processing of elements $k = 0$ and $k = N/2$, because they are stored in a different way from the rest of the elements (see Equation (2)). It also does not contain the code for processing of elements $k = 1$ through $k =$ kTILE-1. These elements are processed separately because they do not comprise a full "tile", and processing them together with the rest of the tiles would require protection with branches, which would ruin vectorization performance. For the complete code listing, refer to the source code available at [1].

4. Using the EFFT Library

EFFT is a C++ library that implements `class EFFT_Transform`:

```
class EFFT_Transform {

// Private members not shown

public:
  EFFT_Transform( // Constructor
        const long n, const int splits);
  ~EFFT_Transform(); // Destructor

  void RunTransform(); // Main method
  float* Data(); // Accessor to input
  float* Result(); // Accessor to output
};
```

Listing 12: Declaration of `EFFT_Transform`

In order to perform a DFT, an object of type `class EFFT_Transform` must be initialized. Its constructor requires two inputs: size (N) and splits (s). The size argument is the size of the full transform of type `long`. Splits is the number of Radix-2 CT that EFFT applies on the data set. Note that our current implementation of EFFT only accepts sizes that are multiples of $2^{(s+8)}$. This restriction is the consequence of the Radix-2 CT algorithm, as well as the tiling optimization we have implemented.

The intialization of an `EFFT_Transform` object may take some time, especially for large arrays. However, once it has been created, it can be reused as many times as the user needs. As discussed in Section (3), the `EFFT_Transform` internally allocates the memory space it needs, including the data array.

After initialization of the main class, the user populates the data array using the accessor `EFFT_Transform::Data()`. Note that this is a pointer to an array of type `float`; EFFT currently does not support double precision.

After the array was populated with data, the user should simply call the method `EFFT_Transform::RunTransform()` to carry out the DFT on the data array. The pointer to the result of the DFT can be obtained with the `EFFT_Transform::Result()` accessor method. The current implementation of EFFT does the Transform out-of-place, so the data array will retain the input and `EFFT_Transform::Result()` and `EFFT_Transform::Data()` will return different pointers.

Listing (13) shows a sample implementation of a 1D DFT using the EFFT library.

```
EFFT_Transform myTransform(n, numofsplits);

//Getting the pointer to the data
float* A = myTransform.Data();

//Populating with random data
srand(0);
for (long int i = 0; i < n; i++) {
  A[i] = ((float)rand()/(float)RAND_MAX)-0.5f;
}

//Applies FFT to the data
myTransform.RunTransform();

//Getting the pointer to the result
float* B = myTransform.Result();
```

Listing 13: Using the EFFT library.

Number of splits, s, is one of the two tuning parameters available in EFFT. Generally, the good performance is achieved when 2^s is nearest to the number of total available threads. The other tuning parameter, number of workers, is set externally by either setting the environment variable `CILK_NWORKERS` or invoking `__cilkrts_set_param("nworkers","Wkrs")` where `Wkrs` is the number of workers to use. For example, in order to use 8 workers, either set `export CILK_NWORKERS=8` in the terminal or invoke `__cilkrts_set_param("nworkers","8")` in the C++ code. Generally, the good performance is achieved when the number of workers is equal to the physical core limit. The optimal value for the two tuning parameter depends on variety of factors, such as the model of the CPU, the performance of the memory subsystem, and the DFT size. The process of this one-time optimization is discussed further in Section (5.2).

5. Benchmarks

5.1. System Configuration

All of the benchmarks presented in this section were taken on a Colfax ProEdge™ SXP8600 workstation based on two-way Intel Xeon E5-2697 v2 processor (12 cores per socket, 24 cores total). We used the Intel C++ compiler version 15.0.0.90 and Intel MKL version 11.2 on a CentOS 6.5 Linux OS. For comparing with FFTW, we used FFTW version 3.3.4 compiled with the Intel C compiler with configuration arguments `--enable-avx`, `--enable-single` and `--enable-openmp`.

5.2. Parameter Tuning

As discussed in Section (4), the tuning of EFFT is done by scanning the 2D parameter space of the number of splits, s, and the number of Intel Cilk Plus workers. Fortunately, this optimization scan is a one-time requirement; the optimal values for the tuning parameters will not change for a given DFT size and hardware. Additionally, the optimization scan is not a requirement to use the EFFT library; it is an option for users who needs the best possible performance. EFFT package comes with a simple benchmarking code which can be used to scan the parameter space.

Figure (5) shows an example of this parameter space scan. Color represents the performance, in GFLOP/s, for each point in the parameter space for a DFT of size $N = 2^{30}$. In this case, the optimal number of splits is $s = 4$ (corresponding to $b = 2^s = 16$ bins), and the optimal number of workers is 16. Here and elsewhere, performance is calculated from the wall clock run time of method `EFFT_Transform::RunTransform()` using Equation (7).

Figure (6) shows the optimal number of workers and splits for EFFT on our system as a function of array size. We use the data shown in this figure to obtain tuned performance measurements in Section 5.3.

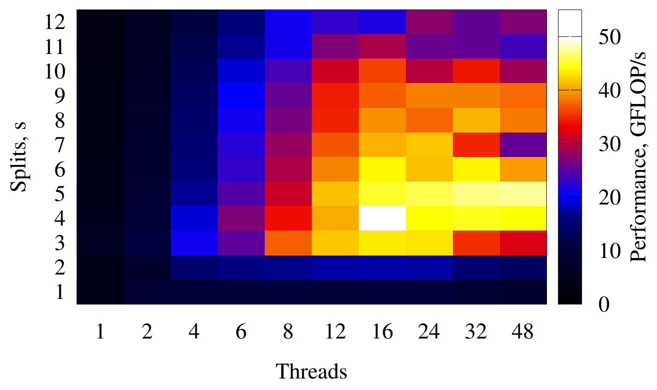

Figure 5: Heat map showing the optimal number of workers and splits vs size. The areas that are "Hot" have better performance.

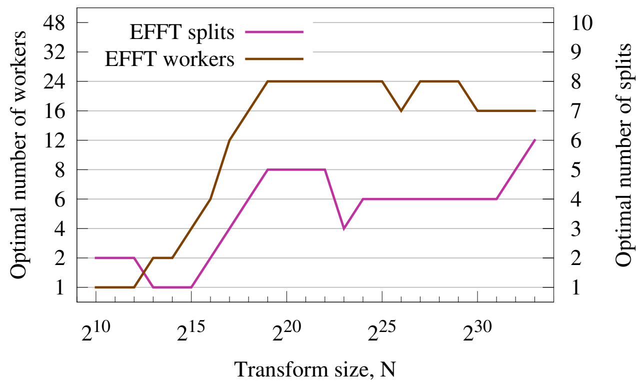

Figure 6: The optimal number of workers and splits vs size. They are different for each size, and thus the parameter scan must be performed for every size to achieve the optimal performance.

In Section 5.3, in addition to EFFT results, we show results of multi-threaded MKL DFT implementation and of the multi-threaded FFTW library. We tuned both of these libraries by tuning the number of threads and thread affinity for each array size. In most cases, 16 or 24 threads and `KMP_AFFINITY=scatter` yielded the best performance. For FFTW, we planned the transforms in the `FFTW_MEASURE` mode and subsequently re-used the "wisdom" generated in these measurements (see [8] for more information on FFTW planning and wisdom).

5.3. Performance, Memory Usage and Accuracy

Figure (7) shows the tuned performance, in GFLOP/s, of EFFT (this work), MKL DFT and FFTW as a function of the array size. The numbers of threads for MKL and FFTW are fixed at, respectively, 16 and 24, which corresponds to values yielding the best results for most array sizes. For EFFT, the number of threads and splits vary from point to point in this plot, and are always set to the optimum value. For FFTW, we did not measure performance for arrays greater than $N = 2^{30}$ because the planning time in the `FFTW_MEASURE` mode is very long (hours to days).

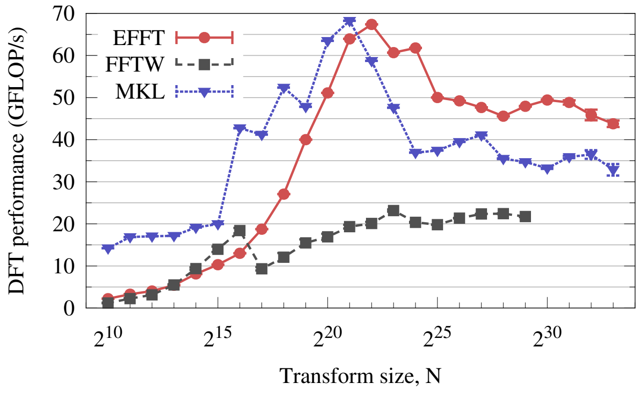

Figure 7: Comparison of performance of our implementation (EFFT), Intel MKL and FFTW, as a function of array size, for $N = 2^q$. The number of threads, T, for MKL is set to the optimal value for each point ($T = 16$ for most points), and for FFTW, $T = 24$ at all points. EFFT performance is reported with the optimal values of tuning parameters T and s at each point.

According to our measurements (Figure 7), EFFT outperforms FFTW for all sizes by more than 2x. It also performs better than MKL for array sizes $N > 2^{22}$, achieving around performance between 45 and 55 GFLOP/s, which is 1.1x to 1.5x faster than MKL.

Figure 8 shows the peak virtual memory usage for each of the libraries. Memory usage was determined by querying the file `/proc/(pid)/status` and reading the line beginning with `VmPeak`. Here, `(pid)` is the process ID of the DFT implementation, and sampling rate was set at 1 kHz. For small array sizes, measurements were not consistent from run to run due to short run times. Solid black line in Figure 8 shows the amount of data in the input DFT array.

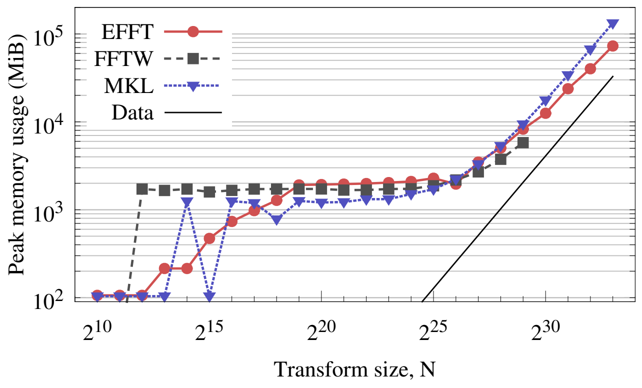

Figure 8: Peak memory usage as a function of array size. We found that for most sizes the memory consumption of EFFT is comparable to memory consumption of MKL DFT.

Figure (8) shows that for small array sizes, all libraries have memory usage around 2 GB. For larger arrays ($N > 2^{28}$), EFFT uses less memory than MKL, however, both libraries require 3x to 4x as much memory as the input data array contains. FFTW has consistently lower memory consumption than EFFT for large arrays, around 2x the data size.

Finally, in order to ensure fair comparison of the three implementations, we estimate the accuracy of the transforms. This is done using the methodology proposed by the FFTW project [8]. The data array is initialized with random values from -0.5 to 0.5 and transformed. Then the computed result, F, is compared to the "exact" solution F^e obtained using a single-threaded "infinite precision" calculation with the help of the GNU Multiple Precision (GMP) library. The comparison metric is the so-called L2 norm of the difference between the computed and "exact" DFT. Equation (9) defines the L2 norm.

$$||F - F^e|| = \frac{\sqrt{\sum(F_k - F_k^e)^2}}{\sqrt{\sum\left(F_k^e\right)^2}} \tag{9}$$

Figure 9 reports the L2 norm of the deviation from the "exact" solution for power of 2 transform sizes with EFFT, MKL and FFTW.

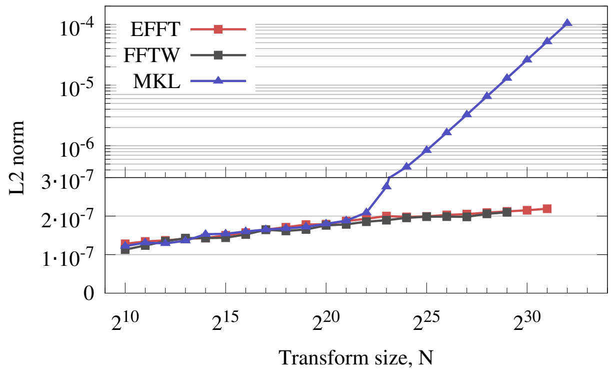

Figure 9: Accuracy of DFTs as a function of array size.

All codes produce comparable L2 norms of order a few times 10^{-7}, except for MKL for large transforms. Starting at the size $N = 2^{22}$, the accuracy of MKL degrades and continues to degrade linearly towards greater sizes. Considering this aspect, EFFT built on small-size single-threaded MKL transforms with Intel Cilk Plus results in both better performance and better accuracy than the multi-threaded MKL DFT implementation.

5.4. Parallel Scalability

While raw maximum performance reported in Section 5.3 is important for practical application, it is also informative to study the parallel efficiency of the implementations. We define parallel efficiency, η, as the ratio of the actual performance with T threads, $P(T)$, to the projected performance assuming linear speedup:

$$\eta(T) = \frac{P(T)}{T \times P(1)}. \tag{10}$$

In compute-bound workloads, η may remain constant up to values of T equal to the number of physical cores in the processor. However, for bandwidth-bound workloads like DFFT, η is expected to decrease with increasing T. In order to evaluate the thread scalability of EFFT, we compare η for EFFT with η for the STREAM benchmark [15].

Figure 10 reports the performance and parallel efficiency of EFFT and of the STREAM benchmark. For EFFT, $N = 2^{30}$ was used. In all tests, thread affinity with T threads was set to `KMP_AFFINITY=explicit,proclist=[0-N]`, where $N = T - 1$. This binds threads to individual processor cores in such a way that for $1 < T \leq 12$, threads are placed on successive cores on CPU 0, and for $T > 12$, threads start filling CPU 1. That said, for $T \leq 12$, only one NUMA node is used in our two-way system, and for $T > 12$, two NUMA nodes are used.

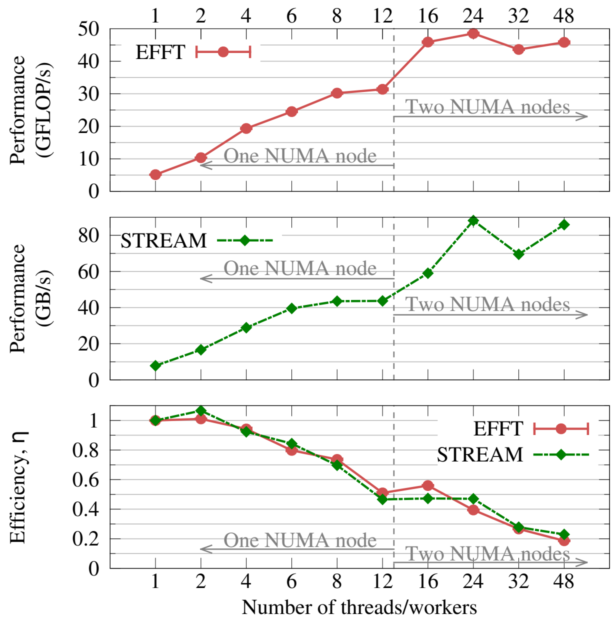

Figure 10: Parallel scalability of EFFT and STREAM.

According to Figure 10, the efficiency of EFFT at the optimal value of $T = 24$ threads is $\eta = 0.39$, while STREAM has better efficiency, $\eta = 0.47$. EFFT is expected to be less efficient than STREAM for $T = 24$. That is because in STREAM, threads read and write only NUMA-local data (i.e., data mapped to memory banks local to the socket on which the thread is executing), while EFFT has a more complex pattern of memory traffic with non-local NUMA accesses. However, for $T = 12$, when NUMA is not involved, EFFT has slightly better efficiency better than STREAM (which is consistent with data reuse in caches in EFFT). This observation indicates good parallel scalability of the bandwidth-bound EFFT algorithm.

6. Discussion

Our goal in this work was to implement a multi-threaded code for very large ($N \gtrsim 10^9$) 1D DFTs. We achieved that goal by developing the EFFT library based on a serial MKL DFT implementation and Intel Cilk Plus to parallelize and vectorize the multi-threaded implementation. Our implementation performs better than multi-threaded MKL DFT for $N > 2^{22}$ by 1.1x to 1.5x, with better accuracy of the calculation and lower memory usage than MKL. We also benchmarked EFFT against FFTW and measured 2x better performance with the former without any loss in accuracy; however, FFTW has lower memory footprint.

This paper also demonstrated multiple optimization techniques and discussed the reasoning behind the performance gain that each technique produced:

- improving temporal data locality via loop tiling,
- improving spatial data locality (in-place algorithm),
- strip-mining to vectorize transcendental math,
- using array notation to vectorize stride-2 operations ("gather" and "scatter"),
- using the `cilk_spawn`/`cilk_sync` extensions of Intel Cilk Plus to effect parallel recursion, and
- using the `cilk_for` extension for loop parallelism.

The optimization methodology presented in this paper is applicable not only to parallel DFFTs, but to a wide variety of computational problems problems.

Futrhermore, this publication demonstrated the strength of Intel Cilk Plus for parallelizing a workload with a complex, multi-level pattern of parallelism. While the OpenMP standard offers similar functionality (tasking and dynamic number of threads in parallel regions), in this application we were not able to achieve satisfactory performance results with OpenMP despite investing a greater development effort than we did with Intel Cilk Plus. In contrast, development with Intel Cilk Plus required little programming effort and resulted in accelerated performance.

The good parallel scalability of EFFT (considering its bandwidth-bound nature) and its reliance on automatic vectorization and portable Intel Cilk Plus parallel framework promise high chances of adapting this application to the Intel MIC architecture. In a future publication, we will report on the possibility of accelerating what we call "enormous Fourier transforms" using Intel Xeon Phi coprocessors.

The product of the publication, the EFFT library, is available for free download [1].

References

[1] Landing page for this paper, 'Intel Cilk Plus for Complex Parallel Algorithms: "Enormous Fast Fourier Transforms" (EFFT) Library'.
http://research.colfaxinternational.com/post/2014/09/18/EFFT.aspx.

[2] Chris Solomon and Toby Breckon. *Fundamentals of Digital Image Processing: A practical approach with examples in Matlab*. John Wiley & Sons, 2011.

[3] W. B. Atwood et al. A Time-differencing Technique for Detecting Radio-quiet Gamma-Ray Pulsars. *The Astrophysical Journal Letters*, 652:L49–L52, November 2006. doi:10.1086/510018.

[4] Matteo Frigo and Steven G Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[5] Jongsoo Park et al. Tera-scale 1D FFT with low-communication algorithm and Intel® Xeon Phi coprocessors. page 34, 2013.

[6] Akira Nukada et al. Scalable multi-gpu 3-d fft for tsubame 2.0 supercomputer. *Proceedings of SC'12*.

[7] Intel Math Kernel Library.
http://software.intel.com/en-us/intel-mkl.

[8] FFTW Library.
http://www.fftw.org/.

[9] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[10] Intel Cilk Plus.
https://software.intel.com/en-us/intel-cilk-plus.

[11] Cilk home page.
http://www.cilkplus.org/.

[12] Michael McCool, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[13] User and Reference Guide for the Intel C++ Compiler 15.0.
https://software.intel.com/en-us/compiler_15.0_ug_c.

[14] Cilk Plus Array Notation for C Accepted into GCC Mainline.
https://software.intel.com/en-us/articles/cilk-plus-array-notation-for-c-accepted-into-gcc-mainline.

[15] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers.
http://www.cs.virginia.edu/stream/.