Large Bandwidth-Efficient FFTs on Multicore and Multi-Socket Systems

Doru Thom Popovici, Tze Meng Low, Franz Franchetti
Department of Electrical and Computer Engineering
Carnegie Mellon University
Email: {dpopovic, lowt, franzf}@cmu.edu

***Abstract*—Current microprocessor trends show a steady increase in the number of cores and/or threads present on the same CPU die. While this increase improves performance for compute-bound applications, the benefits for memory-bound applications are limited. The discrete Fourier transform (DFT) is an example of such a memory-bound application, where increasing the number of cores does not yield a corresponding increase in performance. In this paper, we present an alternate solution for using the increased number of cores/threads available on a typical multicore system. We propose to repurpose some of the cores/threads as soft Direct Memory Access (DMA) engines so that data is moved on and off chip while computation is performed. Overlapping memory accesses with computation permits us to preload and reshape data so that computation is more efficient. We show that despite using fewer cores/threads for computation, our approach improves performance relative to MKL and FFTW by 1.2x to 3x for large multi-dimensional DFTs of up to 2048^3 on one and two-socket Intel and AMD systems.**

I. Introduction

The number of cores on modern shared-memory architectures has been steadily increasing since the introduction of multicore/multi-socket systems around the turn of the century. While the increased number of computational units has benefited compute-bound application such as matrix-matrix multiplication, the performance gain of memory-bound applications such as the Fourier transform (FFT) is still limited. Large Fourier transforms do not use the cache hierarchy and bandwidth to main memory efficiently due to the non-unit-stride access patterns inherent to the algorithm. This makes the task of hiding the latency of strided memory access patterns when accessing main memory difficult.

The inefficient use of the memory hierarchy can be seen in Figure 1, where we compare the results of three parallel implementations against the achievable performance on the Intel Kaby Lake 7700K. We compute the achievable peak performance in Gflop/s assuming data is streamed in and out of the cache hierarchy at full bandwidth speed measured using STREAM [1]. It can be seen that state-of-the-art 3D Fourier transforms offered by MKL 2017.0 and FFTW 3.3.6 achieve at most 47% of the achievable peak when using all eight available threads. In comparison, using various techniques to be discussed in this paper, our parallel implementation yields 80% to 90% of the achievable peak performance.

In this paper, we improve the overall performance of large multidimensional FFTs that do not fit on on-chip cache. Our approach is based on the observation that the FFT is more

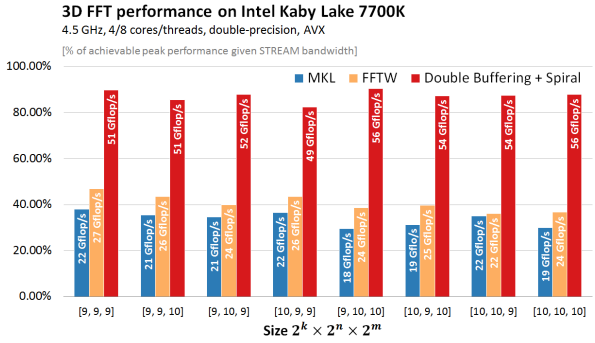

Fig. 1: The plot shows the percentage from achievable peak performance at full bandwidth speed obtained with the STREAM benchmark on the Intel Kaby Lake 7700K. MKL and FFTW achieve at most 47% of achievable peak, while our approach achieves 80% to 90% of peak. The labels on each of the bars represent the unnormalized performance in Gflop/s for the three parallel implementations.

efficient when data is located in the upper levels of the memory hierarchy. We separate data movement and computation to be able to copy data from memory to cache at bandwidth speed in parallel with the computation. In addition, we reshape (pack) data into an appropriate layout to improve performance of subsequent stages. The astute reader may recognize that this approach is similar to the double-buffering approach used in the Out-of-Core (OoC) algorithms such as [2], [3], [4]. The difference is that OoC algorithms read data from disk into main memory, while we buffer the data in the cache hierarchy. Since there are no explicit functions to copy data in the cache similar to the functions used to move data from disk to memory, implementing the double-buffering approach requires less obvious means that are tackled in this paper.

Contributions. The main contributions of this paper are:

- *Systematic way of applying software pipelining.* We show a systematic way of separating and overlapping data movement from memory and computation on cached data for the multi-dimensional Fourier transforms.
- *Double-buffering on cache hierarchies.* We implement a double-buffering mechanism on multicore CPUs. Caches are set-associative and have replacement policies. Full control of data through the cache hierarchy is needed for an efficient double-buffering.

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00048

- *Improve performance of multidimensional FFTs.* We show that our approach improves bandwidth utilization for large Fourier transforms (up to 128 GB of input data) and achieves 1.2x to 3x performance improvements relative to MKL and FFTW on a range of machines from quad-core Haswell/AMD to dual-socket Xeon/Opteron.

II. Background

A. Multicore Architectures

Modern systems ranging from off-the-shelf desktops to high-end servers provide a multitude of features such as large number of threads/cores, large shared cache levels, specialized data movement instructions. All these features are meant to improve performance. However, combining the features to improve memory-bound applications such as the Discrete Fourier transform becomes cumbersome.

Threads and cores. Most modern CPUs offer an increased number of threads and cores to improve parallel execution. Depending on the vendor, threads/cores share different components such as the instruction pipelines and/or the cache hierarchy. This suggests possible locations where threads may contend for resources. Threads executing the same instruction mix will issue instructions to the same execution pipeline. This will cause threads to stall and not make forward progress. Threads with different memory access patterns may conflict within the cache hierarchy and evict each others data. Evicting data to the lower levels of the memory will increase memory access latency if the data is required for future computation.

Cache hierarchy. Caches are data storage that are smaller and faster than main memory located on the same-die as the compute units. Figure 2 shows two typical cache hierarchies for Intel and AMD CPUs. Threads share the different levels of the cache hiearchy. Each cache level is characterized by size and set-associativity. The set-associativy is used to improve cache utilization. Since caches have finite storage, replacement policies are enforced to automatically evict data when other data points are required for computation. This hides explicit data movement from the developers at the cost of reduced control of where data resides.

Non-temporal load and store operations. Loading and storing data from and to main memory is done with temporal and non-temporal load/store instructions. Prefetches may be used, however they are only hints. Of interest to this work are the non-temporal loads and stores. While their temporal counterparts store data in all the cache levels, non-temporal instructions move data "directly" to registers, with the goal of reducing cache pollution. The downside of non-temporal operations appears when data is frequently reused. Since data is not stored in the upper levels of the cache hierarchy, it must always be retrieved from memory, incurring higher latencies.

B. Discrete Fourier Transform

The discrete-time Fourier transform of n input samples $x_0, x_1, \ldots, x_{n-1}$ is described as a matrix-vector multiplication $y = DFT_n x$, where

$$DFT_n = [\omega_n^{kl}]_{0 \le k,l < n}, \text{ where } \omega_n = e^{-j\frac{2\pi}{n}}.$$

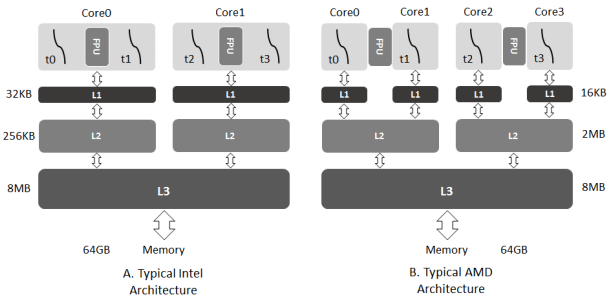

Fig. 2: Two architectures targeted in this paper. Figure A is typical for Intel architectures, where each core has two hardware threads that share the L1 and L2 caches. All cores share L3. Figure B is typical for AMD architectures, where each core has one thread per core with its own L1 data cache. Each two cores share an L2, while L3 is shared by all cores.

The dense matrix DFT_n is the 1D linear transform of size n. Similarly, the multidimensional DFT (MDFT) can be viewed as a matrix-vector multiplication, i.e. $y = DFT_{n_0 \times n_1 \times \ldots \times n_k} x$, where the matrix $DFT_{n_0 \times n_1 \times \ldots \times n_k}$ represents the multi-dimensional FFT.

Computing the DFT or MDFT as a matrix-vector multiplication incurs a $O(n^2)$ arithmetic complexity. Fast Fourier transform algorithms, such as the Cooley-Tukey algorithm, reduce complexity to $O(n \log(n))$. This reduces computation complexity, however the non-unit-stride access patterns inherent to the algorithm persist. Over the past years, there have been numerous works on improving performance for multidimensional FFTs through optimizing data movement. These works can be classified into two main categories.

Reducing memory round-trips. Access to main memory is costly due to the long latency. Reducing the number of round-trips to main memory reduces the overall time incurred by data movement. P3DFFT [5] reorganizes the computation of the 3D FFT by fusing the first and second stages of the 3D FFT when the number of processors is small. This decomposition is known as the slab-pencil decomposition and reduces the number of round trips to main memory from three to two. Johnson et al. [6] proposed the dimensionless FFT where the middle stages are decomposed using the Cooley-Tukey algorithm and are fused with the neighboring compute stages. Similarly, this method reduces the number of round-trips to main memory from three to two. Takahashi et al. [7] proposed a 1.5D decomposition of the parallel 3D FFT implementation. These implementations are orthogonal to our approach since they focus on merging stages, while we are focusing on the row-column algorithm to achieve better streaming behaviour.

Improving data access through layout transforms. Another class of optimizations for multidimensional FFTs focuses on improving memory accesses and data movement through the cache hierarchy by data layout reorganization. Recall that all stages of the MDFT require non-unit-stride memory accesses. Akin et al. [8] have proposed FFT implementations for FPGAs that perform blocked data layout transformations to improve

Matrix formula	Matlab pseudo code
$y = (A_n B_n)x$	`t[0:1:n-1] = B(x[0:1:n-1]);` `y[0:1:n-1] = A(t[0:1:n-1]);`
$y = (I_m \otimes B_n)x$	`for (i=0; i<m; i++)` `  y[i*n:1:i*n+n-1] = B*x[i*n:1:i*n+n-1];`
$y = (A_m \otimes I_n)x$	`for (i=0; i<n; i++)` `  y[i:n:i+m*n-n] = A*x[i:n:i+m*n-n];`
$y = D_n^{mn}x$	`for (i=0; i<n; i++)` `  y[i] = Dn[i, i]*x[i];`

$y = L_m^{mn}x$	`for (i=0; i<m; i++)`
	`  for (j=0; j<n; j++)`
	`  y[i+m*j] = x[n*i+j];`

$y = (L_m^{mn} \otimes I_k)x$	`for (i=0; i<m; i++)`
	`  for (j=0; j<n; j++)`
	`  y[k*(i+m*j):1:k*(i+m*j)+k-1] =`
	`  x[k*(n*i+j):1:k*(n*i+j)+k-1];`

TABLE I: From matrix formulas to code, in Matlab notation.

memory accesses for multidimensional FFTs. Inda et al. [9] have proposed a simple implementation of the FFT using the BSP model where data is reshaped to improve communication between threads. Yzelman et al. [10] proposed a C library that separates data movement from computation, however it does overlapping tasks. Frigo et al. [11] proposed buffering data by copying it to the upper levels of the cache hierarchy before the FFT computation. In all the above papers data is copied before each compute stage. Tang et al. [12] proposed utilizing multiple threads on the Xeon Phi to copy data into the L1 cache and then apply smaller 1D FFTs on it. Each thread executes data movement and computation, however all four threads overlap operations. Song et al. [13] apply data and computation overlapping in the context of large scale 3D FFTs implemented with asynchronous MPI calls. Our work is similar however we target multicore/multi-socket systems with large amounts of main memory, so called fat memory nodes.

C. Kronecker Product Formalism

In this paper we use the matrix-vector representation of the discrete Fourier transform (DFT). We briefly present the Signal Processing Language (SPL) described in [14]. The language is used to capture the implementation of fast algorithms for DFTs as matrix factorizations.

Matrix formalism and SPL. The implementation of fast algorithms for the DFT and MDFT is obtained by factorizing the dense matrix into a product of structured sparse matrices. The decomposition of the DFT and MDFT is captured by the SPL language, which is a mathematical description based on matrix factorization. The language captures the data-flow of the algorithm at a higher level of abstraction. As the backbone of this language, the Kronecker product or tensor product is defined as follows:

$$A \otimes B = [a_{k,l}B], \text{ for } A = [a_{k,l}].$$

The decomposition of the DFT and MDFT revolves around two constructs, namely $I_m \otimes B_n$ and $A_m \otimes I_n$. The first construct applies the matrix B_n on m contiguous blocks of size n, while the latter applies the matrix A_m on m data points located at a stride distance of n elements.

In addition to the Kronecker product, SPL offers constructs such as diagonal and permutation matrices. Diagonal matrices, denoted by D_n^{mn}, are used to specify scaling operations. Permutations and transpositions are used to reshape data. The basic permutation is represented by the L matrix, defined

$$L_n^{mn} : in + j \rightarrow jm + i, 0 \leq i < m, 0 \leq j < n.$$

If the input data is viewed as a matrix of size $m \times n$, then applying the construct L_n^{mn} on the input will produce the transpose matrix of size $n \times m$. Combining the L matrix with the identity matrix I through the Kronecker product allows one to specify block permutations/transpositions, where the block size is given by the size of the identity matrix.

SPL introduces generalizations for the identity matrix I_n

$$I_{m \times n} = \begin{bmatrix} I_n \\ O_{m-n \times n} \end{bmatrix}, \quad m \geq n,$$
$$I_{m \times n} = \begin{bmatrix} I_m & O_{m \times n-m} \end{bmatrix}, \quad m < n.$$

$O_{m \times n}$ is the $m \times n$ all-zero matrix. $I_{n \times n}$ is the identity matrix I_n. Moreover, there are various identities such as

$$A_m \otimes B_n = L_m^{mn}\big(B_n \otimes A_m\big)L_n^{mn} \quad \text{and} \quad L_m^{mn}L_n^{mn} = I_{mn}.$$

that connect the various constructs. All of the constructs can be translated into code following Table I. More on the identities and implementation details can be found in [15], [16].

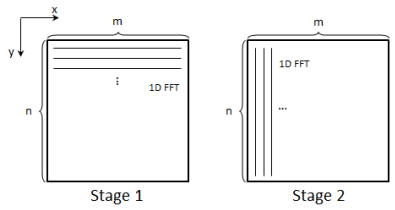

Fig. 3: The implementation of the 2D FFT algorithm. Data is viewed as 2D matrix of size $n \times m$, with the x-dimension corresponding to size m laid out in the fast dimension in memory. The first stage applies 1D FFTs in rows, while the second stage applies the pencils in columns.

D. Fast Implementations for DFT and MDFT

1D DFT. Fast Fourier transform (FFT) algorithms for both DFT and MDFT can be expressed using the SPL language [17]. For example, the well-known Cooley-Tukey algorithm that recursively decomposes a 1D DFT of size $N = mn$ into smaller sub-problems of size m and n can be expressed as follows

$$DFT_{mn} = \big(DFT_m \otimes I_n\big)D_n^{mn}\big(I_m \otimes DFT_n\big)L_m^{mn}.$$

Decomposing the algorithm using the SPL language allows one to visualize the algorithm's data-flow. The algorithm first transposes the data using the L matrix, applies the DFT_n on contiguous blocks of data, scales the result by the twiddle factors and finally applies the DFT_m on strided data points.

Since the focus of this paper is on fast implementations of the 2D and 3D DFTs, we recommend the reader the following

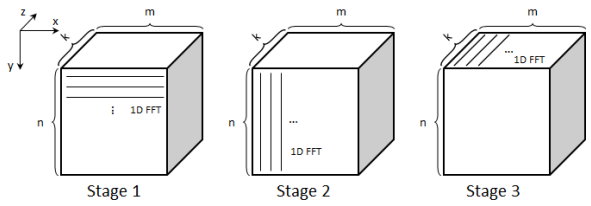

Fig. 4: The implementation of the 3D FFT algorithm. Data is viewed as a 3D cube of size $k \times n \times m$, with the x-dimension corresponding to size m laid out in the fast dimension in memory. The 1D FFTs are applied in all three dimensions.

papers [18], [19] for more details on efficient implementations of the 1D FFT. The main take-away of this section is that FFT algorithms are strided algorithms that suffer from low bandwidth utilization and cache conflicts.

2D and 3D DFT. Similar to the 1D FFT, the multidimensional FFTs are captured by the SPL notation. We start with the 2D FFT and expand to the 3D FFT. The 2D FFT is expressed as a dense matrix $DFT_{n\times m}$ that can be decomposed using the Kronecker as

$$DFT_{n\times m} = DFT_n \otimes DFT_m.$$

Using the general property that multidimensional DFTs are separable operations, the 2D DFT can be further decomposed

$$DFT_{n\times m} = \underbrace{\left(DFT_n \otimes I_m\right)}_{\text{Stage 2}} \underbrace{\left(I_n \otimes DFT_m\right)}_{\text{Stage 1}}.$$

The overall operation described mathematically above is depicted in Figure 3. The data is viewed as a $n \times m$ matrix stored in row-major order. The first stage of the 2D FFT applies 1D FFTs of size m in the rows, whereas the second stage applies 1D FFTs of size n in the columns direction. The above decomposition of the 2D FFT is called the pencil-pencil decomposition, where each pencil refers to a 1D FFT applied in each dimension. Recall that the 1D FFTs require strided access, therefore for large 2D FFTs each of the pencils will require data to be accessed strided from memory.

The same approach can be applied to decompose the 3D FFT, where $DFT_{k\times n\times m}$ represents the dense matrix.

$$DFT_{n\times m} = \underbrace{\left(DFT_k \otimes I_{mn}\right)}_{\text{Stage 3}} \underbrace{\left(I_k \otimes DFT_n \otimes I_m\right)}_{\text{Stage 2}} \underbrace{\left(I_{kn} \otimes DFT_m\right)}_{\text{Stage 1}}.$$

Data is viewed as a 3D cube of size $k \times n \times m$, stored in row major order with the x-dimension corresponding to the size m laid out in the fastest memory dimension. The 3D FFT applies 1D FFTs in each of the three dimensions. The problem of accessing data at strides remains.

III. Overlapping Data Movement with Computation

The goal of our approach is to overlap data movement with computation so that data can be streamed from memory while computation is performed on data previously stored in cache. This requires a re-evaluation of the method the multidimensional FFTs are computed and most importantly how data access can be separated from computation.

A. 2D and 3D FFT Revisited

In this paper, we focus on optimizing data movement for the 2D and 3D FFT. Recall that in each stage, 1D FFTs are applied in each dimension. Some dimensions require the 1D FFTs to be applied at large strides. To reduce those strides and thus have better memory access patterns, data can be reshaped after each compute stage. For the 2D FFT, this translates into a transposition along the main diagonal using an L matrix after each stage of computation expressed as

$$DFT_{n\times m} = \underbrace{L^{mn}_n\left(I_m \otimes DFT_n\right)}_{\text{Stage 2}} \underbrace{L^{mn}_m\left(I_n \otimes DFT_m\right)}_{\text{Stage 1}}.$$

We change the transposition from an element-wise transposition to a blocked transposition as

$$\begin{aligned} DFT_{n\times m} =& \left(L^{mn/\mu}_n \otimes I_\mu\right)\left(I_{m/\mu} \otimes DFT_n \otimes I_\mu\right) && \text{Stage 2} \\ & \left(L^{mn/\mu}_{m/\mu} \otimes I_\mu\right)\left(I_n \otimes DFT_m\right) && \text{Stage 1.} \end{aligned}$$

The $\otimes I_\mu$ produces memory access in cacheline size μ packets [20]. This modification offers benefits when implementing data movement with SIMD instructions. Moreover, it reduces the false sharing when threads are applied for parallelism.

The data reshape technique can be extended to the 3D FFT implementation. However, the transposition in three dimensions becomes a rotation along the data cube's main diagonal. Similar to the 2D FFT, the rotation is meant to assist memory accesses in subsequent stages. The definition of the rotation matrix $K^{k,n}_m$ is

$$K^{k,n}_m = \left(L^{mk}_m \otimes I_n\right)\left(I_k \otimes L^{mn}_m\right).$$

It can be seen that the K matrix has two parts. The first part transposes the front xy-plane. The second part transposes the side xz-plane. The rotation is depicted in Figure 5. This implementation is an element-wise rotation. Therefore, for the same reasons discussed for the 2D case, we block the rotation to move entire cachelines. The adopted 3D FFT decomposition is represented as

$$\begin{aligned} DFT_{k\times n\times m} =& \left(K^{n,m/\mu}_{k\mu} \otimes I_\mu\right)\left(I_{nm/\mu} \otimes DFT_k \otimes I_\mu\right) && \text{Stage 3} \\ & \left(K^{m/\mu,k}_{n\mu} \otimes I_\mu\right)\left(I_{mk/\mu} \otimes DFT_n \otimes I_\mu\right) && \text{Stage 2} \\ & \left(K^{k,n}_{m/\mu} \otimes I_\mu\right)\left(I_{kn} \otimes DFT_m\right) && \text{Stage 1.} \end{aligned}$$

For the remainder of this section, we discuss overlapping computation and communication for the first stage. Same steps are applied to the other stages.

B. Data Movement From Memory vs. Cache

Recall that we want to overlap data movement from memory with computation on cached data. This means that it is necessary to separate main memory data movement from cache data movement. Since all stages of a multidimensional FFT are

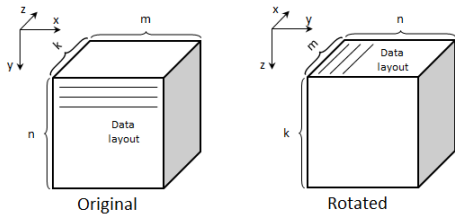

Fig. 5: 3D rotation applied on the data cube after each compute stage. The original data of size $k \times n \times m$ is rotated to a cube of size $m \times k \times n$. The data in the xy-plane is moved to the data points in the yx-plane in the rotated data cube.

similar, we will illustrate the separation of the two types of data movements using the first stage of the 3D FFT.

Working on cached data. The first step towards applying our approach is tiling the computation such that the data required for computation fits in the cache. This means that instead of applying all 1D FFTs, we apply a smaller batch

$$\big(K^{k,n}_{m/\mu} \otimes I_\mu\big) \underbrace{\big(I_{kn} \otimes DFT_m\big)}_{\text{block by } b} = \big(K^{k,n}_{m/\mu} \otimes I_\mu\big)\big(I_{knm/b} \otimes \underbrace{\overline{\big(I_{b/m} \otimes DFT_m\big)}}_{\text{Compute}}\big).$$

The batch size is determined by the size of the shared buffer. If b represents the size of the buffer, then b/m represents the total number of 1D pencils that can be applied, where m represents the size of the 1D FFT.

Each knm/b iteration applies a batch of 1D FFTs on the buffer of size b. We denote the $I_{b/m} \otimes DFT_m$ construct as the compute kernel. We further decorate the compute kernel with $\overline{(\cdot)}$ to specify that the computation must be done inplace, i.e. the input is overwritten with the computed result. The computation kernel is parallelized across the threads assigned for computation or compute-threads.

Identifying main memory access. Data needs to be read from memory and stored into the local buffer before computation can start. Once computation has completed data needs to be stored to main memory from the local buffer. The computation kernel determines the data movement. The SPL notation helps us capture the read/write operations. By reformulating the SPL formula, we obtain the following:

$$\big(K^{k,n}_{m/\mu} \otimes I_\mu\big)\big(I_{knm/b} \otimes \overline{\big(I_{b/m} \otimes DFT_m\big)}\big) = I_{knm/b} \otimes \big(\underbrace{W_{b,i}}_{\text{Store}} \underbrace{\overline{\big(I_{b/m} \otimes DFT_m\big)}}_{\text{Compute}} \underbrace{R_{b,i}}_{\text{Load}}\big).$$

Let $W_{b,i}$ and $R_{b,i}$ denote the data movement matrices that read and write blocks of size b from and to main memory every i iteration. The variable i takes values in $0, knm/b - 1$.

We define two additional constructs $S_{n,b,i}$ and $G_{n,b,i}$ to help with the construction of the read and write matrices. The two constructs are rectangular matrices obtained by vertically/horizontally stacking all-zero matrices $O_{u\times v}$ of various sizes and the identity matrix I of size b.

$$S_{n,b,i} = \begin{bmatrix} O_{ib\times b} \\ I_b \\ O_{(n/b-i-1)b\times b} \end{bmatrix} \text{ where } S_{n,b,i} \in \mathcal{R}^{n\times b}$$

The $G_{n,b,i}$ matrix is the transposed version of $S_{n,b,i}$. To better understand the meaning behind the two constructs, they can be viewed as sliding windows that read/write blocks of size b elements from the input/output. If we consider the I_n as a copy operation of n elements, then the two matrices are slices through the columns and rows of the identity matrix,

$$I_n = \begin{bmatrix} S_{n,b,0} & S_{n,b,1} \dots & S_{n,b,n-1} \end{bmatrix} = \begin{bmatrix} G_{n,b,0} \\ G_{n,b,1} \\ \vdots \\ G_{n,b,n-1} \end{bmatrix}.$$

Combining the two constructs with the K matrix we can define the data movement matrices $W_{b,i} = \big(K^{k,n}_{m/\mu} \otimes I_\mu\big) S_{knm,b,i}$ and $R_{b,i} = G_{knm,b,i} I_{knm}$.

We separate computation from data movement to/from off-chip memory. This creates three dependent tasks as seen in Figure 6. The *Load* task moves b contiguous elements from the input to the cached buffer. The *Compute* task applies the b/m 1D FFTs of size m inplace. The *Store* task copies data back to main memory from the cached buffer once computation finished. The tasks are parallelized across the available threads.

C. Task Parallelization and Scheduling

The SPL notation also allows us to identify which components need to be parallelized. Recall that computation must be performed on data stored in the cached buffer. This implies that the parallelism must be applied on the $I_{b/m} \otimes DFT_m$ construct which will modify the first child as follows

$$I_{knm/b} \otimes \underbrace{\big(W_{b,i}\overline{\big(I_{b/m} \otimes DFT_m\big)}R_{b,i}\big)}_{\text{parallel on } p_d,\, p_c} = I_{knm/b} \otimes \big(\underbrace{W_{b,i}}_{p_d} \underbrace{\big(I_{p_c} \otimes \overline{\big(I_{b/(mp_c)} \otimes DFT_m\big)}\big)}_{p_c} \underbrace{R_{b,i}}_{p_d}\big).$$

Given p_c threads for computation, each thread will apply its $I_{b/p_c m} \otimes DFT_m$ on its own disjoint data points.

Data movement is parallelized across p_d threads. The matrix $R_{b,i}$ copies contiguous blocks of size b from main memory to the cached buffer. Since data is contiguous, data is streamed in. The matrix $W_{b,i}$ writes blocks equal to the size of the cacheline from the cached buffer back to main memory. Since data is rotated and thus written at strides, bandwidth utilization may drop. The write matrix is also parallelized across p_d threads.

We finally apply software pipelining [21] to the outermost construct represented by $I_{knm/b}$. Software pipelining allows the skewing of the *Load*, *Compute* and *Store* tasks and permits the tasks to be executed in parallel. We group the *Load* and *Store* tasks into one task, with the observation that the store operation must precede the load operation. Table II shows

Iteration	Store and Load with p_d threads		Compute with p_c threads	
$i = 0$		$\mathtt{t[i \mod 2]} = \mathtt{R_{b,i}x}$		Prologue
$i = 1$		$\mathtt{t[i \mod 2]} = \mathtt{R_{b,i}x}$	$\mathtt{t[(i+1) \mod 2]} = \big(\mathtt{I_{b/m}} \otimes \mathtt{DFT_m}\big)\mathtt{t[(i+1) \mod 2]}$	
$i = 2$	$\mathtt{y} = \mathtt{W_{b,i-2}t[i \mod 2]}$	$\mathtt{t[i \mod 2]} = \mathtt{R_{b,i}x}$	$\mathtt{t[(i+1) \mod 2]} = \big(\mathtt{I_{b/m}} \otimes \mathtt{DFT_m}\big)\mathtt{t[(i+1) \mod 2]}$	
...		...	...	Steady
$i = knm/b-1$	$\mathtt{y} = \mathtt{W_{b,i-2}t[i \mod 2]}$	$\mathtt{t[i \mod 2]} = \mathtt{R_{b,i}x}$	$\mathtt{t[(i+1) \mod 2]} = \big(\mathtt{I_{b/m}} \otimes \mathtt{DFT_m}\big)\mathtt{t[(i+1) \mod 2]}$	State
$i = knm/b$	$\mathtt{y} = \mathtt{W_{b,i-2}t[i \mod 2]}$		$\mathtt{t[(i+1) \mod 2]} = \big(\mathtt{I_{b/m}} \otimes \mathtt{DFT_m}\big)\mathtt{t[(i+1) \mod 2]}$	Epilogue
$i = knm/b+1$	$\mathtt{y} = \mathtt{W_{b,i-2}t[i \mod 2]}$			

TABLE II: Applying software pipelining to the outer loop of the construct $I_{knm/b} \otimes \big(W_{b,i}\overline{(I_{b/m} \otimes DFT_m)}R_{b,i}\big)$.

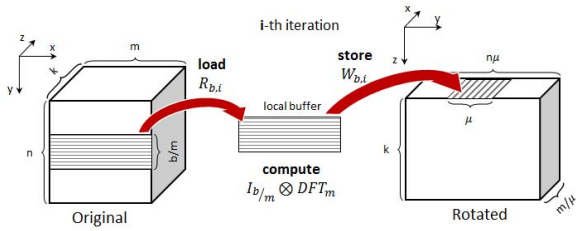

Fig. 6: Separating data movement from computation for the first stage of the 3D FFT. Data is streamed into the local buffer. Computation applies batches of 1D FFTs inplace. Data is rotated/transposed back to main memory.

the prologue, steady state and epilogue of the SPL construct. The prologue loads data into the shared buffer and signals the threads to start computation. The epilogue stores data back to main memory once computation has been completed. In steady state data movement and computation is executed in parallel. While the compute threads apply computation on one half of the shared buffer, the data threads store and load data to and from main memory into the other half of the buffer. Given p the total number of threads, this implies that $p = p_c + p_d$, where p_c and p_d represent the compute and data threads.

D. Parallel Framework and Code Generation

The above mathematical descriptions are translated into code. We first construct a general C code template for doing the double-buffering mechanism. We parallelize the framework using OpenMP [22], more precisely we use `#pragma omp parallel region`, since it gives better control over the threads. Based on the thread ID returned by `omp_get_thread_num`, we determine which threads do data movement and which do computation. Half the threads are used for data movement and half the threads are used for computation. We use `kmp_affinity` on the Intel architectures and `sched_setaffinity` on the AMD architectures to pin the threads to the specific cores and we explicitly use them within the C code. In addition, we use `#pragma omp barrier` to synchronize the threads.

The SPL notation described in this work used to capture the computation and data movement is implemented within the SPIRAL system [15], which is a framework for automatically generating code for linear transforms. We use SPIRAL to automatically generate the computation and data movement using either AVX or SSE, depending on the underlying architecture we target. We parameterize the generated code by the thread ID and the socket ID, since each thread must do its own task. We copy the generated code within the template framework in order to get the full 2D and 3D FFT implementation.

IV. Mitigating Interference

Threads/cores share resources such as execution pipeline, cache hierarchy, main memory and links between multiple sockets. Threads/cores contend for these resources. Usually, interference on the shared resources causes slower execution. In this section, we focus on the main causes of interference and provide solutions to reduce the effects of possible conflicts and thus increase overall performance.

A. Single Socket Execution

Both Intel and AMD offer multiple threads that share the same floating point functional units and have private and/or shared caches. Pinning the threads to specific cores influences the overall performance of the application. We discuss the interference at the functional units and cache level and show solutions to reduce contention.

Interference in the execution pipeline. Irrespective of vendor, threads share the floating point functional units. Therefore, in our approach we group one data-thread and one compute-thread. The threads are pinned together to the same core to share the functional units. Data-threads only load and store data. Compute-threads execute some load and store operations, however they predominantly execute computation such additions/multiplications. On most architectures load/store instructions and arithmetic instructions are issued to different pipelines. Choosing threads with the same instruction mix is not recommended since the threads will conflict for the same execution pipeline. We use `NOP` instructions interleaved within the data-threads task to allow the compute-threads to issue their loads and make progress. Data-threads issue only load/store operations, thus the threads may fully occupy the load/store pipeline. Even though compute-threads have fewer load/store operations, they still require some for computation.

Interference at the cache hierarchy. Pinning a data-thread and a compute-thread so that they share the functional units, makes the threads share some of the cache levels. For example, on the Intel architecture, the two threads/hyperthreads on the cores share both the L1 and L2 cache. In addition to the different mixes of instructions, the two type of threads also have different memory access patterns. Data-threads stream through while reading and rotate the data on the write back. However, the FFT accesses data at strides. The different access patterns causes cache evictions.

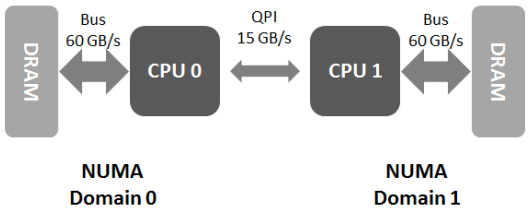

Fig. 7: Two socket system with two NUMA domains. Each NUMA domain consists of one CPU and local main memory. The two processors are connected via QPI(Intel) or HT(AMD).

Non-temporal loads and stores. Recall that non-temporal loads and stores bypass the cache hierarchy to reduce cache pollution. However, not all loads and stores within the code must be non-temporal. In our approach, only the $W_{b,i}$ and $R_{b,i}$ matrices must utilize the non-temporal operations, since those are the only operations that move data to and from memory. The matrix $R_{b,i}$ must read data non-temporally, however it must store the data temporally in the shared buffer. The compute-threads must apply the FFT on the data in the subsequent iteration. However, the matrix $W_{b,i}$ can read and write non-temporally since the computed data is not required until the next FFT stage. The write matrix non-temporally stores cacheline blocks at large strides in main memory.

Cache aware FFT. Computation is cacheline aware, since the FFT is computed at cacheline granularity. Recall that a 1D FFT accesses data at non-unit strides. Accessing data at large strides may cause data to be placed in the same set and thus evict other cachelines before the remaining data within the cacheline is fully consumed. We use SIMD instruction such as SSE and AVX to implement the computation. We follow the details from [18], where the FFT performs a data-format change of the complex data storage between compute stages. The format change swaps from complex interleaved, where the real and imaginary components are interleaved in memory, to a block interleaved format, where blocks of the real components and blocks of the imaginary components are consecutive in memory. This format change is meant to make computation more efficient. Separating the real and imaginary components and using AVX instructions allows computation to be done at cacheline granularity. Since the 2D and 3D FFTs have multiple stages, the format change is applied once in the first stage, the rest of the computation is done in block interleaved and in the last stage data is changed to complex interleaved. This change of format is different from the one presented in [9] which is meant to improve communication between threads.

Cache aware buffer allocation. The cached buffer t shared by the data-threads and the compute-threads must reside within the cache hierarchy. More precisely, the buffer must be located in the last level cache (LLC) since it is shared between all the threads. We set the size of the buffer to be equal with half of the LLC, $b = \text{size}_{\text{LLC}}/2$. The buffer cannot fully occupy the LLC since computation also requires extra temporaries for storing partial results and constants such as the twiddle factors.

B. Dual Socket Execution

We extend our approach to two socket systems, where each socket belongs to a Non-Uniform Memory Access (NUMA) domain. Each NUMA domain has private main memory. The sockets can access their memory through fast bandwidth buses and they can access neighbor's main memory through data-links, such as Intel's QuickPath Interconnect (QPI) or AMD's HyperTransport (HT). Figure 7 shows the topology of a typical two socket system. It is worthwhile noticing that bandwidth to main memory within a NUMA domain is higher compared to the bandwidth over the data links. The difference in bandwidth suggests computation to be done on data stored locally and transfers over the interconnects to be kept to a minimum.

We only extend the 3D FFT to two sockets. The expression

$$\underbrace{I_{knm/b} \otimes \big(W_{b,i}\big(I_{b/m} \otimes DFT_m\big)R_{b,i}\big)}_{\text{parallel on } sk,\ p_c,\ p_d} = I_{sk} \otimes \Big(\underbrace{I_{knm/bsk} \otimes \big(W_{b,i,sk}\big(I_{b/m} \otimes DFT_m\big)R_{b,i,sk}\big)}_{\text{parallel on } p_c,\ p_d}\Big)$$

specifies the order in which we parallelize the first stage of the 3D FFT, the same steps are taken for the other two stages. We parallelize the construct first over the number of sockets sk and then over the number of data-threads p_d and compute-threads p_c. The parallelism over the sockets modifies the read and write matrices, since all data points are required for the computation of the 3D FFT. Thus, data needs to be exchanged across the data links.

Since communication over the QPI/HT links is costly, we apply a slab-pencil decomposition to the 3D FFT, where the first two stages communicate only within the NUMA domain. We split the data-set in the z-dimension, where each socket receives a contiguous block of size $\big(k/\text{sockets}\big) \times n \times m$. Each node can compute a 2D FFT locally and transpose locally, without crossing the interconnect. In order to compute the 1D FFT in the z-dimension data needs to be exchanged across the data-links. Another communication over the interconnect is needed once the 1D pencil is fully computed so that data is put in the correct order. Figure 8 depicts the data movement in the three compute stages of the 3D FFT. Table III presents the generalized versions of write $W_{b,i}$ matrices. All three matrices are parameterized by the number of sockets k. By setting the number of sockets equal to $\mathrm{sk} = 1$, the implementation defaults to the single-socket implementation. Reading data is done by each socket from the local memory and has the same representation as the single-socket implementation.

V. Experimental Results

Experimental setup. We now evaluate the performance of our approach. We run experiments on Intel and AMD systems with one or two sockets. All systems provide multiple cores/threads that share a large last level cache. All Intel architectures have hyperthreads enabled. AMD does not offer hyperthread support. For the dual-socket systems we configure the QPI/HT protocol to use Home Snoop. The Home Snoop

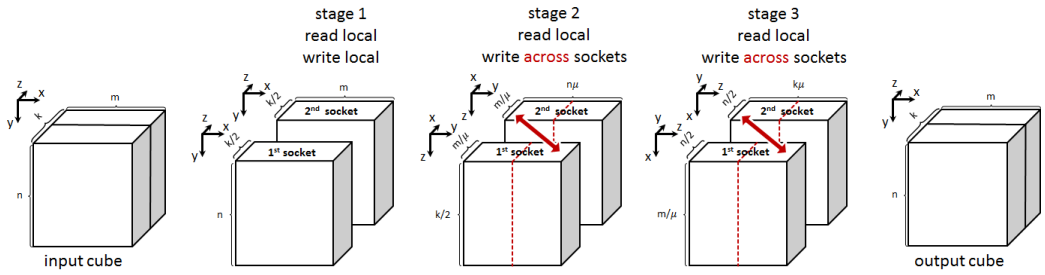

Fig. 8: The three different data cube shapes after each 3D FFT stage. Data is distributed across the k-dimension to each socket. The first stage reads and writes the data locally, while the other two stages read data locally but write data across the sockets.

Matrix	SPL Representation
$W^1_{i,b,sk}$	$= (I_{sk} \otimes K^{n,k/sk}_{m/\mu} \otimes I_\mu) S_{knm,b,i}$
$W^2_{i,b,sk}$	$= (L^{sknm/\mu}_{nm/\mu} \otimes I_{k\mu/sk})(I_{sk} \otimes K^{k/sk,m/\mu}_{n} \otimes I_\mu) S_{knm,b,i}$
$W^3_{i,b,sk}$	$= (L^{skk}_{k} \otimes I_{mn/sk})(I_{sk} \otimes K^{m/\mu,n/sk}_{k} \otimes I_\mu) S_{knm,b,i}$

TABLE III: The SPL representations for the three write matrices (rotation matrices) applied after each compute stage.

protocol improves bandwidth over the interconnect since it reduces cache traffic used for cache coherence.

1) One socket systems: Intel Haswell 4770K, Intel Kaby Lake 7700K and AMD FX-8350 - 8 threads, 8 MB L3 cache, 32/64/64 GB DRAM, bandwidth 20/40/12 GB/s
2) Two socket systems: Intel Haswell 2667v3, AMD 6276 Interlagos (Bluewaters): 16 threads, 20/16 MB L3 cache, 256/64 GB DRAM, bandwidth 85/20 GB/s

We compare our implementation to MKL 2017.0 and FFTW 3.3.6 on the Intel architectures. On the AMD systems we compare our implementation only against FFTW 3.3.6. All libraries are compiled with OpenMP, AVX and SSE enabled. On the Intel architectures, we use `MKL_DYNAMIC`, `MKL_NUM_THREADS` and `KMP_AFFINITY` to control the number of threads and the placement of the threads within the MKL library. On the AMD architectures, we use `OMP_NUM_THREADS` and `GOMP_AFFINITY` to control the threads within the FFTW library. We compile all code with the ICC compiler version 2017.0 and the GCC compiler 4.8.5-20150623. All code is compiled with the `-O3` flag. We do not compare the current implementation against the SPIRAL generated parallel code presented in [20]. The previous parallel implementation targeted medium size 1D FFTs and did not offer support for compute/communication overlap.

Performance metric. We report performance for our approach, MKL and FFTW as billion floating point operations divided by runtime in seconds. We use $5N\log(N)$ as estimate for the flop count which over-estimates the number of operations. The resulting Pseudo-Gflop/s is proportional to inverse runtime and is an accepted performance metric for the FFT. We use the `rdtsc` time-stamp counter and the CPU frequency to compute the overall execution time.

We also compare performance of all three parallel implementations against the performance when streaming data in and out of cache at bandwidth speed. For the upper bound we consider infinite compute resources and do not take computation time into account. We use the STREAM benchmark [1] to determine the achievable bandwidth (**GB/s**) for each targeted architecture. We determine the performance when streaming the total amount of data as

$$P_{\text{io}} = \frac{5 \cdot N \cdot \log(N) \cdot \text{bandwidth}_{\text{STREAM}}}{2 \cdot N \cdot \text{nr}_{\text{stages}} \cdot \text{sizeof(double)}},$$

where $\text{nr}_{\text{stages}}$ represents the number of compute stages in the FFT, N represents the size of the FFT and sizeof(double) represents the size of a double precision floating point in GB. The current implementation offers support for complex numbers therefore the total size is multiplied by two.

2D FFT. We first present results for the 2D FFT implementation. Figure 9 shows the results of the three implementations compared to the achievable peak when streaming data at 40 GB/s. Our double-buffering approach achieves on average 74% of the achievable peak. However, MKL and FFTW implementation achieve on average 50% of peak. Two aspects are worth noticing. First, for small sizes bandwidth utilization is less than 80% since the number of iterations $\text{iter} = mn/b$ in each compute stage is small. For example, $\text{iter} = 4$ when $b = 131,072$, $m = 512$ and $n = 1,024$. Second, as the size of the 2D FFT increases bandwidth utilization drops. Recall that after each compute stage data needs to be transposed similarly to the rotation described in Figure 6. The transposition is applied on a panel of size $b/m \times m$, where m is the size of the 1D FFT and b is the size of the shared buffer. As m increases, b/m decreases, therefore TLB misses cannot be amortized. We leave as future work other methods of separating data movement from computation for cases where the size of the 1D FFT is equal or greater than the size of the shared buffer.

3D FFT. We show results for the 3D FFT on multiple architectures and compare them against the achievable peak. We evaluate large 3D FFTs that do not fit on the cache. Figure 1 shows the results on the Intel Kaby Lake 7700K. Our implementation achieves 80% to 90% of practical peak, whereas MKL and FFTW achieve at most 47%. Our approach uses the bandwidth and the cache hierarchy more efficiently and therefore outperforms MKL and FFTW by almost 3x. It is important to state that the 3D FFT does not experience the problems displayed by the 2D FFT. First, the number of iterations in each compute stage is higher. For example,

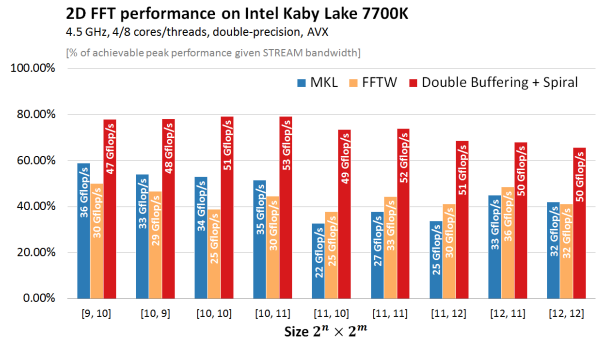

Fig. 9: The plot shows performance of the 2D FFT on the Intel Kaby Lake 7700K. We compare against the achievable performance when data is streamed at bandwidth speed. Our approach achieves on average 75% ofl peak. The labels on the bars show unnormalized performance.

iter $= 1,024$, when $b = 131,072$, $m = 512$, $n = 512$, $k = 512$. Given the problem sizes tackled in this paper, the size of the 1D FFTs will not influence the data movement. Opening new memory pages can be easily be amortized. The 3D FFT will most likely run out of main memory.

Similar results are obtained on the Intel Haswell 4770K (top left plot in Figure 11), where our implementation achieves on average 30 Gflop/s and almost 2x faster execution compared to MKL and FFTW. Our approach improves bandwidth utilization and the implementation runs at 92% of peak. Due to space constraints we do not present these bandwidth results. The top right plot in Figure 11 shows the results on the AMD FX-8350, where our double-buffering approach improves bandwidth utilization and performance. It is important to notice that the speedup over FFTW on AMD is only 1.6. The reason is that FFTW uses the slab-pencil decomposition for the 3D FFT and this decomposition is suitable for AMD's larger caches.

Extending to two sockets. We finally present results on platforms with two sockets. For all implementation, we use the NUMA library to allocate and partition data on each NUMA node. Figure 10 shows the performance results on the two-socket Intel Haswell 2667. It can be seen that our implementation outperforms those offered by MKL and FFTW. However, the 3D FFT implementation only achives 1.2x to 1.6x performance improvements. The performance drop is caused by the communication over the QPI link. Recall that the double-buffering 3D FFT requires two write operations over the QPI link as described in Figure 8. Writing data over the interconnect is expensive compared to reading data, therefore our implementation is penalized. Assuming that data is streamed at bandwidth speed and that the QPI link does not influence the transfer rate, than our implementation is within 20% to 30% of achievable peak. However, given the data movement presented in Figure 8, where two data movements over the QPI are required, our double buffering 3D FFT implementation performs within 7% to 15% from the performance given the cummulative bandwidth speed of the main memory and QPI links.

We further present scaling results keeping the problem size fixed and increasing the number of sockets from one to two. The bottom two plots in Figure 11 show the results on the two-socket Intel and AMD architecture. For the Intel architecture, given the data movement in Figure 8 our approach improves performance on average by 1.7x when increasing the number of sockets. Communication over the QPI link and conflicts between the two types of threads limit the overall improvement. On the AMD system the HT link runs at a similar bandwidth speed as the bus to main memory, therefore the slowdown caused by the interconnect is smaller. We do not report comparison results against FFTW for the AMD two-socket system since the FFTW library misbehaves on the Bluewaters system and provides buggy results.

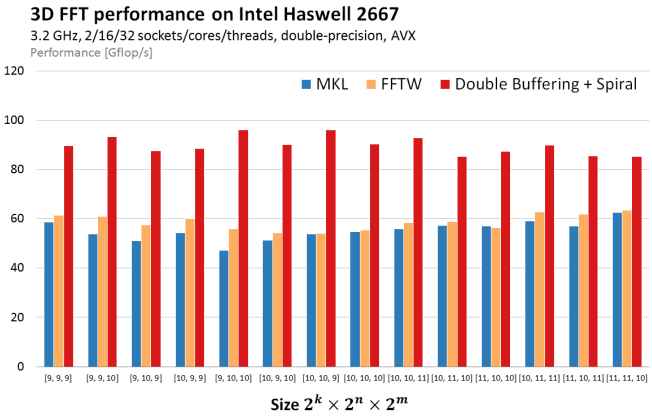

Fig. 10: The plot shows performance for the 3D FFT on the two-socket Intel Haswell 2667. We report Gflop/s and show that our implementation outperforms MKL and FFTW on two socket systems given the data movement in Figure 8.

VI. Conclusion

In this paper, we make the observation that large FFTs that do not fit in cache under-utilize the cache hierarchy and the bandwidth to main memory. By repurposing some of the compute-threads as soft DMA engines to load and reshape data in parallel to computation our approach streams data from memory at near peak bandwidth speed. We show that despite using less threads for computation our approach improves overall bandwidth utilization to within 10% of peak bandwidth on single socket systems and within 30% on dual-socket systems. Our implementation improves performance relative to MKL and FFTW by 1.2x to 3x on single and dual-socket systems for problem sizes that have a memory footprint of up to 128 GB and do not fit on modern GPUs.

Acknowledgment

This work was sponsored partly by the DARPA PERFECT and BRASS programs under agreements HR0011-13-2-0007 and FA8750-16-2-003, and NSF through award ACI 1550486. The content, views and conclusions presented in this document are those of the author(s) and do not necessarily reflect the position or the policy of the sponsoring agencies.

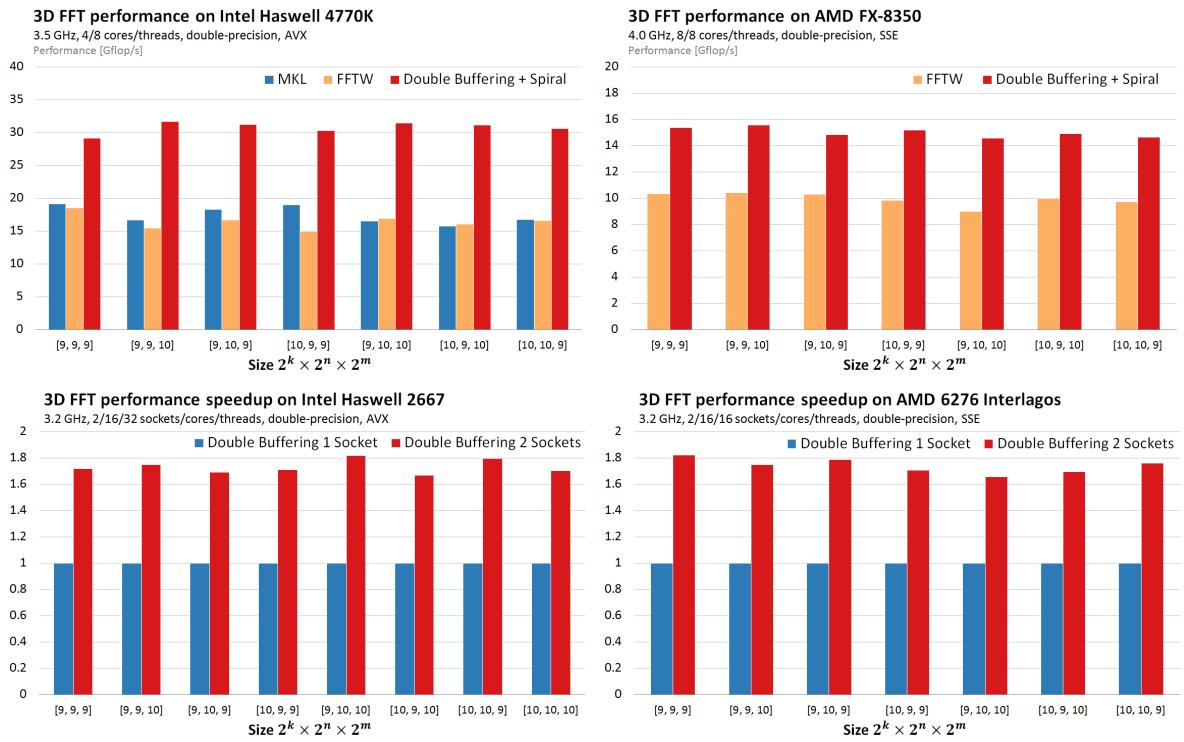

Fig. 11: The top plots present the performance in Gflop/s for the 3D FFT implementation on the Haswell and AMD architecture. The bottom plots show speedup for large 3D FFTs when increasing the number of sockets for fixed problem sizes.

REFERENCES

[1] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, 1995.

[2] D. H. Bailey, "FFTs in external of hierarchical memory," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, pp. 234–242, ACM, 1989.

[3] W. M. Gentleman and G. Sande, "Fast Fourier Transforms: For fun and Profit," in *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pp. 563–578, ACM, 1966.

[4] T. H. Cormen, J. Wegmann, and D. M. Nicol, "Multiprocessor out-of-core FFTs with distributed memory and parallel disks (extended abstract)," IOPADS '97, (New York, NY, USA), pp. 68–78, ACM, 1997.

[5] D. Pekurovsky, "P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C192–C209, 2012.

[6] J. Johnson and X. Xu, "A recursive implementation of the dimensionless FFT," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2003.

[7] D. Takahashi, A. Yee, T. Hoefler, C. Coti, J. Kim, and F. Cappello, "An implementation of parallel 3-D FFT with 1.5-d decomposition," in *The seventh workshop of the INRIA-Illinois-ANL Joint Laboratory on Petascale Computing*, 2012.

[8] B. Akin, F. Franchetti, and J. C. Hoe, "FFTs with near-optimal memory access through block data layouts: Algorithm, architecture and design automation," *Journal of Signal Processing Systems*, 2015.

[9] M. A. Inda and R. H. Bisseling, "A simple and efficient parallel FFT algorithm using the BSP model," *Parallel Computing*, vol. 27, no. 14, pp. 1847–1878, 2001.

[10] A. Yzelman, R. H. Bisseling, D. Roose, and K. Meerbergen, "Multicorebsp for c: a high-performance library for shared-memory parallel programming," *International Journal of Parallel Programming*, vol. 42, no. 4, pp. 619–642, 2014.

[11] S. G. Johnson and M. Frigo, "Implementing FFTs in practice," in *Fast Fourier Transforms* (C. S. Burrus, ed.), ch. 11, Rice University, Houston TX: Connexions, September 2008.

[12] J. Park, G. Bikshandi, K. Vaidyanathan, P. T. P. Tang, P. Dubey, and D. Kim, "Tera-scale 1D FFT with low-communication algorithm and the Intel Xeon Phi coprocessors," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pp. 34:1–34:12, ACM, 2013.

[13] S. Song and J. K. Hollingsworth, "Designing and auto-tuning parallel 3-D FFT for computation-communication overlap," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pp. 181–192, ACM, 2014.

[14] J. Xiong, J. Johnson, R. W. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Programming Languages Design and Implementation (PLDI)*, pp. 298–308, 2001.

[15] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[16] F. Franchetti, Y. Voronenko, and M. Püschel, "Formal loop merging for signal transforms," in *Programming Languages Design and Implementation (PLDI)*, pp. 315–326, 2005.

[17] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*. Philadelphia, PA, USA: SIAM, 1992.

[18] D. T. Popovici, F. Franchetti, and T. M. Low, "Mixed data layout kernels for vectorized complex arithmetic," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.

[19] M. Frigo, "A fast Fourier transform compiler," in *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, vol. 34, pp. 169–180, ACM, May 1999.

[20] F. Franchetti, Y. Voronenko, and M. Püschel, "FFT program generation for shared memory: SMP and multicore," in *Supercomputing (SC)*, 2006.

[21] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," pp. 318–328, 1988.

[22] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," May 2015.