FFTS WITH NEAR-OPTIMAL MEMORY ACCESS THROUGH BLOCK DATA LAYOUTS

Berkin Akın, Franz Franchetti and James C. Hoe

Carnegie Mellon University
Department of Electrical and Computer Engineering
Pittsburgh, PA, USA

ABSTRACT

Fast Fourier transform algorithms on large data sets achieve poor performance on various platforms because of the inefficient strided memory access patterns. These inefficient access patterns need to be reshaped to achieve high performance implementations. In this paper we formally restructure 1D, 2D and 3D FFTs targeting a generic machine model with a two-level memory hierarchy requiring block data transfers, and derive memory access pattern efficient algorithms using custom block data layouts. Using the Kronecker product formalism, we integrate our optimizations into Spiral framework. In our evaluations we demonstrate that Spiral generated hardware designs achieve close to theoretical peak performance of the targeted platform and offer significant speed-up (up to 6.5x) compared to naive baseline algorithms.

Index Terms— Discrete Fourier transforms, fast Fourier transforms, algorithm design and analysis, data layout

1. INTRODUCTION

Scientific and digital signal processing applications require single and multidimensional fast Fourier transforms (FFTs) on large size and high precision data sets. For these problem sizes, the FFT computation proceeds as stages by constantly transferring small portions of the data set to and from the external memory (e.g. DRAM or disk). However, well-known algorithms using standard data layout schemes require large strided memory accesses which leads to poor performance on many target platforms.

In this work we develop 1D, 2D and 3D FFT algorithms using block data layout schemes targeting a generic high level machine model which captures the major memory characteristics of various platforms. Tuning the large size FFT algorithms for block data layout schemes, namely tiled and cubic data layouts, allows reshaping the inefficient strided access patterns, which is the key problem in achieving high performance. *Our goal is to derive restructured FFT algorithms that transfer large contiguous blocks of data throughout the computation.*

We use the Kronecker product formalism [1] to formally represent and manipulate the FFT algorithms. Formal representation allows us to compactly capture the complicated data permutations and manipulate them to derive custom data layout FFT algorithms. Further it enables us to abstract algorithmic choices and the machine model in the same formal framework. Using the formal representation, we integrate our optimizations into the hardware and software synthesis framework Spiral [2]. Our hardware based evaluations show that the optimized designs generated by Spiral can achieve close to theoretical peak performance and offer significant speed-up compared to naive baseline algorithms.

This work was sponsored by DARPA under agreement HR0011-13-2-0007. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA.

Related work. There have been many implementations of single and multidimensional FFTs on various platforms. These include software implementations on CPUs [3, 4], GPUs [5], supercomputers [6, 7], and hardware implementations [8, 9, 10]. These implementations either do not address the memory access pattern issue or provide a solution for a specific target platform and problem. In this work, we address this issue for 1D, 2D and 3D FFTs targeting a generic machine model which captures the major memory characteristics of various platforms.

Organization. Section 2 provides the background on the targeted machine model, FFT and Kronecker product formalism. Next, Section 3 discusses FFT algorithms using block data layouts. Then the main contribution of the paper, the formal representation and derivation of these algorithms, is provided in Section 4. Finally, Section 5 evaluates the implementations of the derived algorithms.

2. BACKGROUND

Machine model. In this work we target a high level abstract machine model that has three main components: (1) Main memory and data transfer, (2) local memory, and (3) compute. *Main memory*, represents the S_M-size large but slow storage medium (e.g. DRAM, disk, distributed memory, etc.) which is constructed from smaller S_B-size *data blocks* (e.g. DRAM rows, disk pages, MPI messages, etc.). Accessing an element from a data block within the main memory is generally associated with an initial high latency cost (A_M^{miss}). After the initial access, accessing consecutive elements from the same data block has substantially lower latency (A_M^{hit}) where $A_M^{\text{miss}} = A_M^{\text{hit}} + C$ and C is an overhead cost whose value depends on the particular platform. Hence the initial high latency cost of accessing a data block in the main memory can be best amortized by transferring the whole contiguous chunk of elements of the accessed data block. In contrast, *local memory*, is S_L-size small buffer used for fast access to the local data (e.g. cache, scratchpad, local cluster node, etc.). We assume that local memory can hold multiple data blocks of the main memory i.e. $S_M > S_L > S_B$. Finally, *compute* represents the functional units that actually process the local data (e.g. vector unit, ALU, etc.). Various high-performance and parallel computing platforms ranging from embedded processors up to distributed supercomputers fit into this high-level machine model. FFT algorithms should be carefully fitted to these architectures to achieve high performance and power/energy efficiency.

Fast Fourier Transform. Computing the discrete Fourier transform (DFT) of an n-element input vector corresponds to the matrix-vector multiplication $y = \text{DFT}_n x$, where x and y are n point input and output vectors respectively, and

$$\text{DFT}_n = \left[\omega_n^{k\ell}\right]_{0 \le k,\ell < n}, \quad \omega_n = e^{-2\pi i/n}.$$

978-1-4799-2893-4/14/$31.00 ©2014 IEEE

Computation of the DFT by direct matrix-vector multiplication requires $O(n^2)$ arithmetic operations. Well-known fast Fourier transform (FFT) algorithms reduce the operation count to $O(n \log n)$. Using the Kronecker product formalism in [1], an FFT algorithm can be expressed as a factorization of the dense DFT into product of structured sparse matrices. For example, the well-known Cooley-Tukey FFT [11] can be expressed as

$$\mathrm{DFT}_{nm} = (\mathrm{DFT}_n \otimes \mathrm{I}_m)\, \mathrm{D}_m^{nm} (\mathrm{I}_n \otimes \mathrm{DFT}_m)\, \mathrm{L}_n^{nm}. \quad (1)$$

In (1), L_n^{nm} represents a *stride permutation* matrix that shuffles its input vector x and generates output vector y as

$$x[im + j] \rightarrow y[jn + i], \quad \text{for } 0 \le i < n,\ 0 \le j < m.$$

If x is viewed as an $n \times m$ matrix, stored in row-major order, then L_n^{nm} performs a transposition of this matrix. Further, I_n is the $n \times n$ identity matrix, and $\otimes$ is the *Kronecker*, or *tensor*, product which is defined as

$$A \otimes B = [a_{i,j} B], \quad \text{where } A = [a_{i,j}].$$

Finally, D_m^{nm} is a diagonal matrix of *twiddle factors*.

Multidimensional DFTs can also be considered as simple matrix-vector multiplications, e.g. $y = \mathrm{DFT}_{n \times n \times \ldots \times n}\, x$ where

$$\mathrm{DFT}_{n \times n \times \ldots \times n} = \mathrm{DFT}_n \otimes \mathrm{DFT}_n \otimes \ldots \otimes \mathrm{DFT}_n. \quad (2)$$

Similar to single dimensional DFT, multidimensional DFTs can be computed efficiently using multidimensional FFT algorithms. For example the well-known row-column algorithm for 2D-DFT can be expressed in tensor notation by using (2) and tensor identities [1] as

$$\mathrm{DFT}_{n \times n} = \left(\mathrm{L}_n^{n^2} (\mathrm{I}_n \otimes \mathrm{DFT}_n)\, \mathrm{L}_n^{n^2}\right)\left(\mathrm{I}_n \otimes \mathrm{DFT}_n\right). \quad (3)$$

The overall operation of (3) is demonstrated in Figure 1(a). Here, assuming a standard row-major data layout, the first stage (row FFTs) leads to sequential accesses in main memory, whereas the stride permutations ($\mathrm{L}_n^{n^2}$) in the second stage (column FFTs) correspond to stride-n accesses which is demonstrated in Figure 1(b).

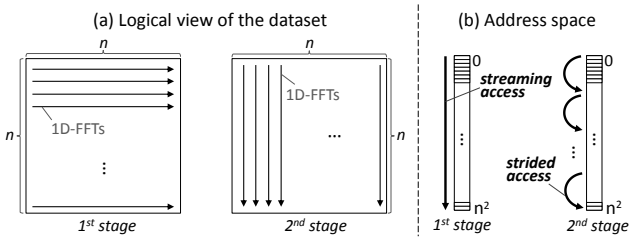


Fig. 1. Overview of row-column 2D-FFT computation in (3).

Similarly, by using (2) and tensor identities in [1] the well-known 3D decomposition algorithm for 3D-DFT can be represented as, where $A^M = M^{\mathrm{T}} A M$,

$$\mathrm{DFT}_{n \times n \times n} = \left(\mathrm{I}_{n^2} \otimes \mathrm{DFT}_n\right)^{\mathrm{L}_{n^2}^{n^3}} \left(\mathrm{I}_{n^2} \otimes \mathrm{DFT}_n\right)^{\mathrm{I}_n \otimes \mathrm{L}_n^{n^2}} \left(\mathrm{I}_{n^2} \otimes \mathrm{DFT}_n\right). \quad (4)$$

The overall operation of (4) is demonstrated in Figure 2. If we assume a sequential data layout of the cube in x-y-z direction, first stage (FFTs in x) corresponds to sequential accesses to main memory however, due to the permutation matrices $\mathrm{I}_n \otimes \mathrm{L}_n^{n^2}$ and $\mathrm{L}_{n^2}^{n^3}$, second and third stages (FFTs in y and z, respectively) require stride n and stride n^2 accesses respectively (very similar to 1(b)).

Until now we discussed decomposing large 2D and 3D-FFTs into small 1D-FFT computation stages that fit in the local memory considering the described machine model. A large 1D-FFT whose data set do not fit in local memory requires similar decomposition into smaller 1D-FFT kernels (e.g. Cooley-Tukey decomposition in (1)). In (1), we observe the same stride permutations as the row-column 2D-FFT algorithm. Hence, from a memory access pattern point of view, overall large size 1D-FFTs are handled very similar to the 2D-FFT computation (see Figure 1).

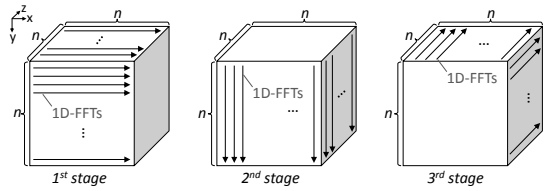


Fig. 2. Overview of 3D-decomposed 3D-FFT computation in (4).

In summary, conventional large size FFT algorithms that use standard data layouts require strided accesses. These strided access patterns correspond to accessing different data blocks in the main memory. Continuously striding over data blocks does not allow amortizing the high latency cost of the main memory accesses, which yields very low data transfer bandwidth and high energy consumption. While there are FFT algorithms like the *vector recursion* [12] that ensure block transfers, they require impractically large local storage for data block sizes dictated by the main memory.

3. FFTS USING BLOCK DATA LAYOUTS

Changing the spatial locality of the memory accesses by adapting a customized block data layout in the main memory enables avoiding inefficient strided access patterns. By avoiding strided accesses and transferring large contiguous blocks of data one can amortize the latency of the main memory accesses and often also a large energy overhead. In this paper we focus on tiled and cubic data layout schemes.

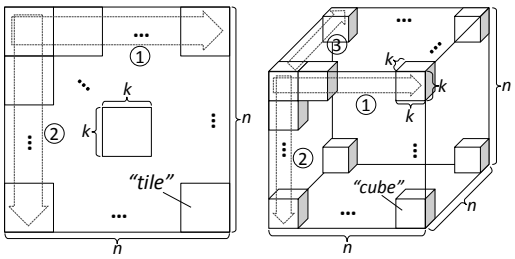
Tiling is basically a block data layout, where n^2 element vectors are considered as $n \times n$ element matrices which are divided into $k \times k$ element small tiles (Figure 3(a)). Then the elements within tiles are mapped to physically contiguous locations in the main memory. When the tile size is selected to match the data block size in the main memory, transferring a tile corresponds to transferring a whole contiguous data block from main memory.

As explained in [8], given the tiled data layout, one can compute the 2D-FFT while avoiding strided accesses. The main idea is instead of transferring stripe of elements in row and column direction as shown in Figure 1, transfer "tiles" in row and column direction (see ① and ② in Figure 3(a)).

From a memory access pattern perspective, 2-stage Cooley-Tukey algorithm for 1D-FFT has the same behavior as the row-column 2D-FFT algorithm as mentioned in Section 2. Hence the tiled data layout can be used in computing the 1D-FFT to avoid strided accesses.

Similarly in the cubic data layout, n^3 element data set is abstracted as a $n \times n \times n$ element three dimensional cube which is divided into $k \times k \times k$ element smaller cubes (see Figure 3(b)). Then each $k \times k \times k$ element small cube is physically mapped into a contiguous data block in main memory. Hence transferring a cube corresponds to transferring a whole data block from the main memory.

Cubic data layout enables 3D-FFT computation without strided accesses. Similar to the 2D-FFT scheme, the main idea is instead of transferring stripe of elements in x, y and z direction as demonstrated in Figure 2, transfer "cubes" in x, y and z direction (see ①, ② and ③ in Figure 3(b)).



(a) Tiled representation. (b) Cubic representation.

Fig. 3. Logical view of the dataset for tiled and cubic representation.

A custom data layout comes along with address translation scheme that maps the logical addresses to the physical locations in the main memory. Formal representation of a custom data layout scheme is given as $\text{DFT} = ((\text{DFT})^{\overleftarrow{Q}})^{\overrightarrow{P}}$ where $Q = P^{-1}$. Here $\overleftarrow{(.)}$ and $\overrightarrow{(.)}$ represent the address translation and the data layout respectively. This representation only makes the data layout and the address mapping constructs explicitly labelled, the overall operation is still a natural DFT computation. P and Q are simply stride permutation matrices. For example, the identity matrix (i.e. $P = Q = \text{I}$) corresponds to the standard sequential data layout. It is also possible to represent block data layouts by the stride permutation matrices.

Although conceptually straightforward, capturing the details of the overall operation algorithmically is non-trivial. The discussion above omits the complexity of the data permutations and the details of the local computation. Formal representation in tensor notation allows capturing all the non-trivial details of the algorithms and the machine model in the same framework. Abstracting the algorithm and the machine model in the same framework allows detailed formula manipulations targeting the machine model, which is crucial to achieve high performance implementations.

Spiral. Spiral is an automated tool for generation and optimization of hardware and software implementations of linear signal transforms including DFT [2, 9]. Spiral takes formula representation of a given transform (e.g. DFT_n), then expands and optimizes it recursively using internal *rewrite rules*, resulting in a structured formula, which is subsequently translated into implementation. The formal representation in tensor notation used in this paper enables integrating our optimizations into Spiral. This way we can generate implementations of optimized FFT and search the alternative design possibilities automatically.

4. FORMALLY RESTRUCTURED ALGORITHMS

In our approach, we identify set of formula identities that restructure the given FFT formula so that it can be mapped to the data layout scheme and target machine model efficiently. The goals for restructuring the FFT are (i) to make sure that all the permutations that correspond to main memory accesses are restructured to transfer tiles/cubes, and (ii) to breakdown the formula constructs such that the local permutations and local 1D-FFT computations fit in the local memory.

Rewrite Rules Rewrite rules are set of formula identities that capture the restructuring of the FFT algorithms. We list the necessary formula identities used in restructuring the algorithms for the tiled data layout in Table 1. The labels on the restructured formula constructs represent the implied functionality in the implementation. $\overleftarrow{(.)}$ and $\overrightarrow{(.)}$ represent the address translation and the data layout respectively. $|$ represents a memory fence. $\text{I}_\ell \bar{\otimes}$ corresponds to iteration operator. Finally, $\underline{(.)}$ and $\overline{(.)}$ correspond to the local kernel (permutation or computation) and the main memory data transfer permutation

Table 1. Tiled mapping rewrite rules.

$$A \rightarrow (A^{\overleftarrow{Q}})^{\overrightarrow{P}}, \quad \text{where } Q = P^{-1} \tag{5}$$

$$AB \rightarrow A|B \tag{6}$$

$$\text{I}_n \otimes A_n \rightarrow \text{I}_{n^2}(\text{I}_{n/k} \bar{\otimes} \underline{\text{I}_k \otimes A_n})\,\text{I}_{n^2} \tag{7}$$

$$A_n \otimes \text{I}_n \rightarrow \text{L}^{n^2}_n(\text{I}_{n/k} \bar{\otimes} \underline{\text{I}_k \otimes A_n})\,\text{L}^{n^2}_n \tag{8}$$

$$\text{I}_{n^2} \rightarrow \underbrace{(\text{I}_{n/k} \bar{\otimes} \underline{\text{L}^n_k \otimes \text{I}_k})}_{R_{9b}}\underbrace{\overline{(\text{I}_{n/k} \otimes \text{L}^n_{n/k} \otimes \text{I}_k)}}_{R_{9a}} \tag{9}$$

$$\text{L}^{n^2}_n \rightarrow \underbrace{(\text{I}_{n/k} \bar{\otimes} \underline{\text{L}^{nk}_k})}_{R_{10b}}\underbrace{(\overline{\text{L}^{n^2/k}_{n/k}} \otimes \text{I}_k)}_{R_{10a}} \tag{10}$$

Table 2. Cubic mapping rules. R_j refers to the right hand side of (j).

$$\text{I}_n \otimes \text{I}_n \otimes A_n \rightarrow \text{I}_{n^3}(\text{I}_{n^2/k^2} \bar{\otimes} \underline{\text{I}_{k^2} \otimes A_n})\,\text{I}_{n^3} \tag{11}$$

$$\text{I}_n \otimes A_n \otimes \text{I}_n \rightarrow (\text{I}_n \otimes \text{L}^{n^2}_n)(\text{I}_{n^2/k^2} \bar{\otimes} \underline{\text{I}_{k^2} \otimes A_n})(\text{I}_n \otimes \text{L}^{n^2}_n) \tag{12}$$

$$A_n \otimes \text{I}_n \otimes \text{I}_n \rightarrow \text{L}^{n^3}_n(\text{I}_{n^2/k^2} \bar{\otimes} \underline{\text{I}_{k^2} \otimes A_n})\,\text{L}^{n^3}_{n^2} \tag{13}$$

$$\text{I}_{n^3} \rightarrow (\text{I}_{n/k} \otimes \text{L}^n_k \otimes \text{I}_{nk})(\text{I}_{n^2/k^2} \bar{\otimes} \underline{\text{L}^{nk}_{k^2} \otimes \text{I}_k})$$
$$\overline{(\text{I}_{n^2/k^2} \otimes \text{L}^n_{n/k} \otimes \text{I}_{k^2})(\text{I}_{n/k} \otimes \text{L}^n_{n/k} \otimes \text{L}^n_{n/k} \otimes \text{I}_k)} = R_{14} \tag{14}$$

$$\text{I}_n \otimes \text{L}^{n^2}_n \rightarrow (\text{I}_{n/k} \otimes \text{L}^n_k \otimes \text{I}_{nk})(\text{I}_{n^2/k^2} \bar{\otimes} \underline{\text{L}^{nk^2}_{k^2}(\text{I}_{n/k} \otimes \text{L}^{k^2}_k \otimes \text{I}_k)})$$
$$\overline{(\text{I}_{n/k} \otimes \text{L}^{n^2}_{n/k} \otimes \text{I}_k)(\text{I}_{n/k} \otimes \text{L}^n_{n/k} \otimes \text{I}_{nk})} = R_{15} \tag{15}$$

$$\text{L}^{n^3}_{n^2} \rightarrow (\text{I}_{n/k} \otimes \text{L}^n_k \otimes \text{I}_{nk})(\text{I}_{n^2/k^2} \bar{\otimes} \underline{\text{L}^{nk^2}_{k^2}})$$
$$\overline{(\text{L}^{n^3/k}_{n^2/k^2} \otimes \text{I}_k)(\text{I}_n \otimes \text{L}^n_k \otimes \text{I}_n)} = R_{16} \tag{16}$$

respectively. These labelled formula constructs are restructured base cases, hence an FFT algorithm that consists only of these constructs considered to be final restructured algorithm.

Tiled 2D-FFT. We now apply the rewrite rules to a given 2D-DFT problem to obtain the restructured FFT. For $\text{DFT}_{n\times n}$, we assume the n^2 element data set size (S_D) do not fit in the local memory, i.e. $S_M > S_D > S_L$, hence the large $\text{DFT}_{n\times n}$ should be decomposed into smaller DFT_n stages as discussed. Further we assume that $k \times k$ tiles match the data block size S_B and local memory can hold a whole stripe of n/k tiles, i.e. $S_L \geq S_B \times n/k$. We now derive a tiled 2D-FFT targeting this machine model. Due to space limitations we briefly mention the derivation steps but rather focus on the structure of the final derived algorithm.

The starting point is $\text{DFT}_{n\times n}$. First, rule (5) defines the data layout and corresponding address mapping. Then $\text{DFT}_{n\times n}$ is expanded into smaller DFT stages by using (3) which are separated via memory fence by (6). Next, rules (7)-(8) make the data permutations explicit and label the kernel computation. Finally, rules (9)-(10) restructure the data permutation so that they correspond to tile transfer operations. The result is given in (17). We observe that the local permutation and computation kernel size, $k \times n$ elements, fit in the local memory. However, derived algorithm is restricted to the problem sizes for which a whole stripe of tiles can be held simultaneously in the local memory so that the kernels can be processed locally (remember $S_L \geq S_B \times n/k$ where $S_B = k^2$ elements). Inspection of the (17) shows that all of the formula constructs are labelled base cases of the tiling rewrite rules, hence this formula corresponds to a final optimized algorithm.

Considering the DFT computation is a matrix multiplication, the

Table 3. Final restructured FFT algorithms: 2D, 3D and 1D respectively. (R_9-R_{16} are given in (9)-(16) in Table 1 & 2.)

$$\mathrm{DFT}_{n\times n} = \Big(\big(R^{\mathrm{T}}_{10a} R^{\mathrm{T}}_{10b} (\mathrm{I}_{n/k} \tilde{\otimes} \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n}) R_{10b} R_{10a} \mid R^{\mathrm{T}}_{9a} R^{\mathrm{T}}_{9b} (\mathrm{I}_{n/k} \tilde{\otimes} \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n}) R_{9b} R_{9a}\big)^{\overleftarrow{Q}}\Big)^{\overrightarrow{P}}, \quad \text{where } P = Q^{-1} = \mathrm{I}_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_k \,. \tag{17}$$

$$\mathrm{DFT}_{n\times n\times n} = \Big(\big((\mathrm{I}_{n^2/k^2} \tilde{\otimes} \underline{\mathrm{I}_{k^2} \otimes \mathrm{DFT}_n})^{R_{16}} \mid (\mathrm{I}_{n^2/k^2} \tilde{\otimes} \underline{\mathrm{I}_{k^2} \otimes \mathrm{DFT}_n})^{R_{15}} \mid (\mathrm{I}_{n^2/k^2} \tilde{\otimes} \underline{\mathrm{I}_{k^2} \otimes \mathrm{DFT}_n})^{R_{14}}\big)^{\overleftarrow{Q}}\Big)^{\overrightarrow{P}}, \; P = Q^{-1} = (\mathrm{I}_{n^2/k^2} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_{k^2})(\mathrm{I}_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_k) \tag{18}$$

$$\mathrm{DFT}_{n^2} = \Big(\big(R^{\mathrm{T}}_{10a} R^{\mathrm{T}}_{10b} (\mathrm{I}_{n/k} \tilde{\otimes} \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n}) R_{10b} R_{10a} \mid R^{\mathrm{T}}_{9a} R^{\mathrm{T}}_{9b} \underline{\mathrm{D}^{n^2}_n} (\mathrm{I}_{n/k} \tilde{\otimes} \underline{\mathrm{I}_k \otimes \mathrm{DFT}_n}) R_{10b} R_{10a}\big)^{\overleftarrow{Q}}\Big)^{\overrightarrow{P}}, \quad \text{where } P = Q^{-1} = \mathrm{I}_{n/k} \otimes \mathrm{L}^n_{n/k} \otimes \mathrm{I}_k \,. \tag{19}$$

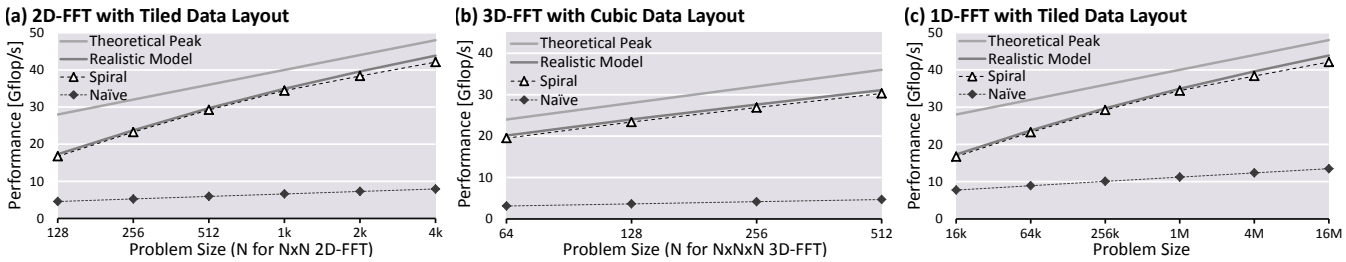


Fig. 4. Performance results of 2D, 3D and 1D-FFTs for theoretical upper bound, realistic model upper bound, Spiral generated and naive baseline hardware implementations on Altera DE4 FPGA. (All single precision floating point)

constructs in the resulting algorithm (17) are performed from right to left on the input data set. First, R_{9a} reads tiles, i.e. whole contiguous data blocks, from the main memory. Then, R_{9b} shuffles the local data to natural order and then 1D-FFTs are applied to the local data. Finally, R^{T}_{9b} re-shuffles the local data after FFT processing and R^{T}_{9a} writes the local data back into the main memory as tiles, which concludes the first stage. The algorithm consists of two stages separated by a memory fence (|). Overall operation in the second stage is very similar to the first stage except the permutations. The second stage has the permutations R_{10a-b} instead of R_{9a-b} where R_{9-10} are given in (9)-(10).

In addition to the tiled 2D-FFT algorithm, we use the set of formula identities (5)-(10) given in Table 1 to derive optimized tiled algorithms for large 1D-FFT (see (19)). Further, we use formula identities (11)-(16) shown in Table 2 to derive 3D-FFT algorithms using cubic data layout (see (18)). The formulas given in (18) and (19) are final optimized algorithms for 3D and 1D FFTs, however due to space limitations we omit the derivations.

This work focuses only on 1D, 2D and 3D FFTs using tiled and cubic data layouts, yet the mathematical framework can easily be extended to higher dimensional FFTs using higher dimensional hypercube data layouts.

5. EVALUATION

We included the formula identities shown in Table 1 and Table 2 into Spiral's formula rewriting system so that Spiral drives the optimized algorithms automatically (e.g. (17), (18) and (19)). In this section we evaluate the 1D, 2D and 3D-FFT designs generated by Spiral.

Algorithms derived in this work can be implemented in hardware or software on various platforms since we target a generic machine model. However, our evaluations are based on hardware implementations on an Altera DE4 FPGA platform. DE4 FPGA platform comes with two channels of total 2 GB DDR2-800 DRAM which corresponds to the main memory considering the machine model described in Section 2, so S_M = 2 GB. DRAM rows are the data blocks in the main memory and the DRAM row buffer size is 8 KB so S_B = 8 KB. Strided accesses to different data blocks (i.e. DRAM rows) yield 1.16 GB/s DRAM data transfer bandwidth whereas transferring contiguous DRAM row buffer size data chunks results in 11.87 GB/s bandwidth out of given 12.8 GB/s theoretical peak. We have measured the penalty of non-contiguous access to a different data block as approximately, $A^{\mathrm{miss}}_M - A^{\mathrm{hit}}_M = C = 20$ clock cycles at 200 MHz. DE4 further provides 2.53 MB of on-chip SRAM which we consider as the local memory, hence S_L = 2.53 MB. Finally, floating point units correspond to the compute element in the machine model. For example, a single precision 4K×4K 2D-FFT has a total data set of S_D = 128 MB where $n = 4096$. Further we need 32×32 single precision complex valued element tiles to match the S_B = 8 KB, so $k = 32$. This overall configuration fits in the described machine model assumptions, i.e. $S_M > S_D > S_L \geq S_B \times n/k$.

Our evaluation results are summarized in Figure 4. We report performance in "Gflop/s" which is calculated as $5n\log_2(n)/t$ for DFT_n where t is the total runtime. Thus higher is better. In Figure 4, we provide (i) bandwidth bounded theoretical peak performance for Altera DE4 where we assume zero latency but limited bandwidth DRAM and infinitely fast on-chip processing, (ii) a realistic peak performance where we include DRAM latency cost and bounded on-chip processing, (iii) actual results from Spiral generated implementations on Altera DE4, and (iv) performance of a naive baseline implementation on the same platform with non-optimized DRAM access patterns. Optimized implementations generated by Spiral always transfer contiguous data blocks from main memory (i.e. whole DRAM rows), whereas the naive implementations have inefficient access patterns that strides over data blocks. Optimized use of the DRAM row buffer leads to efficient DRAM bandwidth utilization, hence Spiral generated implementations offer up to 6.5x higher performance than the naive baseline algorithms by reaching on average 83% of the theoretical peak performance and 97.5% of the realistic peak performance. Further, based on our simulations using DRAMSim2 [13], they achieve 5.5x more energy efficiency in DRAM.

6. CONCLUSION

In this work we formally derive efficient memory access pattern FFT algorithms for large problem sizes using block data layout schemes and targeting a generic machine model which captures the major memory characteristics of various platforms. Formal representation in tensor notation allows us to capture both algorithmic manipulations and the machine model in the same framework. We integrate our optimizations into the Spiral and generate designs for 1D, 2D and 3D-FFTs automatically. Spiral generated designs can achieve close to the theoretical peak performance of the target platform and offer significant speed-up compared to the naive algorithms.

7. REFERENCES

[1] C. Van Loan, *Computational frameworks for the fast Fourier transform*. SIAM, 1992.

[2] 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," *Proc. of IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232–275, 2005.

[3] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura, "Discrete Fourier transform on multicore," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 90–102, 2009.

[4] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. IEEE Intl. Conf. Acoustics Speech and Signal Processing (ICASSP)*, vol. 3, 1998, pp. 1381–1384.

[5] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete Fourier transforms on graphics processors," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC)*, 2008, pp. 2:1–2:12.

[6] M. Eleftheriou, B. Fitch, A. Rayshubskiy, T. J. C. Ward, and R. Germain, "Scalable framework for 3D FFTs on the Blue Gene/L supercomputer: Implementation and early performance measurements," *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 457–464, 2005.

[7] D. Bailey, "FFTs in external or hierarchical memory," in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 1989, pp. 234–242.

[8] B. Akin, P. A. Milder, F. Franchetti, and J. C. Hoe, "Memory bandwidth efficient two-dimensional fast Fourier transform algorithm and implementation for large problem sizes," in *Proc. of the 20th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 188–191.

[9] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Transactions on Design Automation of Electronic Systems*, vol. 17, no. 2, 2012.

[10] Q. Zhu, B. Akin, H. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti, "A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3D Systems Integration Conference (3DIC), 2013 IEEE International*, Oct 2013, pp. 1–7.

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

[12] F. Franchetti and M. Püschel, *Encyclopedia of Parallel Computing*. Springer, 2011, ch. Fast Fourier Transform.

[13] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comp. Arch. Letters*, vol. 10, no. 1, pp. 16–19, 2011.