# Memory Bandwidth Efficient Two-Dimensional Fast Fourier Transform Algorithm and Implementation for Large Problem Sizes

Berkin Akın, Peter A. Milder, Franz Franchetti, James C. Hoe
*Electrical and Computer Engineering Department*
*Carnegie Mellon University, Pittsburgh, PA, USA*
$\{$*bakin, pam, franzf, jhoe*$\}$*@ece.cmu.edu*

***Abstract*—Prevailing VLSI trends point to a growing gap between the scaling of on-chip processing throughput and off-chip memory bandwidth. An efficient use of memory bandwidth must become a first-class design consideration in order to fully utilize the processing capability of highly concurrent processing platforms like FPGAs. In this paper, we present key aspects of this challenge in developing FPGA-based implementations of two-dimensional fast Fourier transform (2D-FFT) where the large datasets must reside off-chip in DRAM. Our scalable implementations address the memory bandwidth bottleneck through both (1) algorithm design to enable efficient DRAM access patterns and (2) datapath design to extract the maximum compute throughput for a given level of memory bandwidth. We present results for double-precision 2D-FFT up to size 2,048-by-2,048. On an Altera DE4 platform our implementation of the 2,048-by-2,048 2D-FFT can achieve over 19.2 Gflop/s from the 12 GByte/s maximum DRAM bandwidth available. The results also show that our FPGA-based implementations of 2D-FFT are more efficient than 2D-FFT running on state-of-the-art CPUs and GPUs in terms of the bandwidth and power efficiency.**

***Keywords*-2D-FFT, 2D-DFT, Memory Bandwidth, DRAM, FPGA.**

## I. Introduction

While there has been extensive prior work in DSP transforms for FPGAs, their designed performance is typically achievable only when the datasets are readily accessed from fast on-chip SRAMs. In this paper, we develop high-performance FPGA implementations of the 2D-FFT for large problem sizes where the datasets must be held externally in DRAM. The calculation must proceed in stages where only a portion of the dataset that fits on-chip is operated on at a time, requiring data elements to make multiple roundtrips to and from DRAM. The constant reading and writing of data elements in DRAM exert a heavy pressure on the available memory bandwidth and is typically the determining factor in the overall performance.

Our development approach combines both careful algorithmic and architecture-level design optimizations. The hardware design effort is preceded by an investigation into DRAM-compatible 2D-FFT algorithms. Standard 2D-FFT algorithms require large strided accesses through the DRAM; their low spatial locality makes very inefficient use of the DRAM row buffer. We find a restructured algorithm with symmetric stages that makes full use of each DRAM row touched, allowing close to theoretical peak DRAM bandwidth to be sustained throughout the calculation.

## II. Background

**Fast Fourier transform.** An $n$ point discrete Fourier transform ($\mathrm{DFT}_n$) is the matrix-vector multiplication:

$$y = \mathrm{DFT}_n\, x, \qquad \mathrm{DFT}_n = [\omega_n^{k\ell}]_{0\le k,\ell<n}, \qquad \omega_n = e^{-2\pi i/n}$$

A literal calculation of $\mathrm{DFT}_n$ by matrix-vector multiplication requires $O(n^2)$ arithmetic operations. So-called *Fast Fourier Transforms* (FFTs) are algorithms that compute $\mathrm{DFT}_n$ in $O(n \log n)$ arithmetic operations.

The two-dimensional DFT (2D-DFT) is again a matrix-vector multiplication. The input and output vectors now have $n^2$ complex elements. These $n^2$-element vectors are usually interpreted as holding a 2D $n$-by-$n$ array in row-major order.

Similar to the 1D-DFT, the 2D-DFT can be computed efficiently using "fast" 2D-FFT algorithms [1]. For example, the well-known row-column algorithm can be summarized as follows: Taking the vector to be a 2D $n$-by-$n$ array, first apply $n$-point 1D-FFT to each of the $n$ rows and then to apply $n$-point 1D-FFT to each of the $n$ columns. The first stage of the calculation accesses the $n^2$-element input vector's elements sequentially; the second stage of the calculation requires stride-$n$ accesses.

**DDR-SDRAM operation.** The *row buffer* is a fast buffer storing the most recently referenced row from a bank of DRAM. High row buffer locality can improve DRAM performance significantly: if the accessed bank and row pair are already active, i.e., the referenced row is already in the row buffer, then a *row buffer hit* occurs which reduces the access latency and energy consumption considerably. On the other hand, when a different row in the active bank is accessed, a *row buffer miss* occurs, requiring the newly referenced row to be read into the row buffer.

On the Altera DE4 platform with DDR2-800 SO-DIMM modules, the maximum bandwidth of 6 GByte/s (per channel) can only be approached when accessing memory in aligned 8KByte (or multiple of) chunks (8KByte is the row buffer size). In stark contrast, large-strided accesses that access only one 8-byte double-precision value per DRAM row yield only 155 MByte/s. Of the two cases above, the

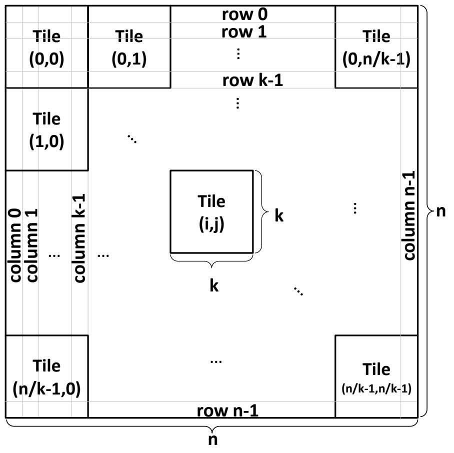


Figure 1. 2D abstraction of dataset.

former is the case for the first stage of the row-column 2D-FFT algorithm; the latter is the case for the second column-wise stage.

## III. DRAM-Friendly Two-Dimensional FFTs

A bandwidth efficient implementation of 2D-FFT algorithm must only interact with DRAM in large row buffer sized chunks.

**Tiled data remapping.** So far, we have assumed the common row-major mapping of the 2D $n$-by-$n$ array onto the $n^2$-element vector. As such, a row-wise traversal of the array results in the efficient sequential memory accesses while a column-wise traversal results in problematic stride-$n$ accesses. To avoid the strided memory access pattern during the column-wise traversal step, we have to alter the spatial locality of memory accesses by choosing a different mapping as illustrated in Figure 1.

We logically divide the $n$-by-$n$ array into $n/k$-by-$n/k$ tiles where each square tile has $k^2$ elements. Instead of the conventional row-major mapping, we map the elements within a *tile* to consecutive locations and we then order the whole tiles in row-major order. The size of a tile ($k^2$) is selected to match the size of the DRAM row buffer. By requiring our desired 2D-FFT implementation to access DRAM only in the granularity of at least a full tile, we ensure efficient DRAM accesses.

**Row-column algorithm with tiled data mapping.** After the remapping, the row-stage progresses in groups of $k$ row-wise 1D-FFTs. To do so, we read each complete row of tiles, and reshuffle their data on-chip into $k$ natural-ordered data rows so that $k$ 1D-FFTs can be applied. After applying the $k$ 1D-FFTs on the rows, we reshuffle the data back into tile-order and write back a row of tiles. Similarly in the column stage, we read each complete column of tiles, apply $k$ 1D-FFTs on the $k$ data columns, then write back the column of tiles. We still perform the same number of reads and writes as the original row-column algorithm, but since we always transfer row buffer sized tiles to and from DRAM, we maximize DRAM bandwidth utilization.

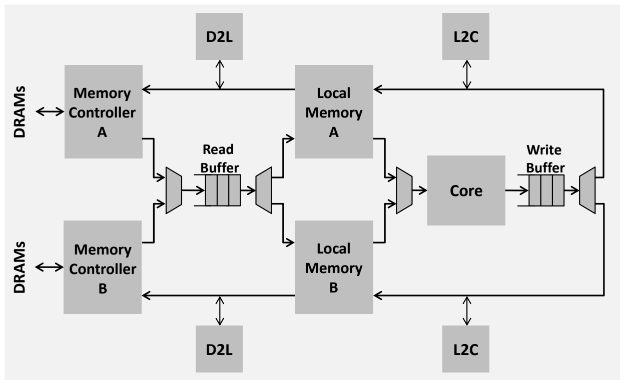


Figure 2. Detailed view of resulting architecture.

**Final algorithm.** Instead of creating a fixed design instance, we built a parametrized design generator that can quickly create designs tuned for different platforms with different row buffer sizes or DRAM bandwidths. Internally, our generator uses a formalism based on tensor products [1] to represent and manipulate the FFT algorithms. Using this formalism, we can further derive a conceptually more challenging but implementation-wise more elegant design, where the operations of the two stages become identical and therefore can reuse the same exact datapath. Without providing a formal derivation, our final 2D-FFT algorithm can be summarized as follows: We bring in each row of tiles and apply 1D-FFT on the $k$ data rows as before. However, when writing back, we locally transpose the contents of each tile and then write back as a column of tiles. By introducing this on-the-fly transposition, stage two becomes identical to stage one and can reuse the same datapath.

## IV. Architecture

In this section, we present our scalable parametrized datapath design that can sustain the maximum DRAM bandwidth throughout the computation based on the algorithm we discussed in the previous section. Figure 2 conceptually depicts the datapath.

**Memory Controllers.** The Altera DE4 platform provides two independent DDR channels. In our usage, a memory controller (Altera's High Performance Controller) attained over 90% of the theoretical peak bandwidth (6 GByte/s for DDR2-800). To make use of the two independent channels, for the first stage of the algorithm, the input vector starts from DRAM on channel A, and the output vector is produced into the channel B. The multiplexers and demultiplexers shown in Figure 2 enable the two DDR channels to exchange their input/output roles for the second stage of the algorithm reusing the same datapath without modification.

**Local Memory.** In both stages of our algorithm, the row-wise applications of the 1D-FFT progress in steps of $k$ rows of the input matrix at a time, corresponding to reading a row of tiles. In each step, the working-set of $n \times k$ elements is buffered in Local Memory constructed from embedded SRAM on the FPGA. Because the data elements are brought in from DRAM in tiled order, the D2L controller in Figure 2 has to re-linearize the data elements into row-order in the Local Memory. To manage this data shuffling without bank conflicts in light of the multiple elements arriving in tile-order per cycle is a non-trivial problem; we construct our solution using the technique given in [2]. To sustain the maximum DRAM bandwidth continuously, we employ double buffering in Local Memories. We can overlap the loading, unloading, and compute operations corresponding to different rows of tiles using this technique.

**Computational Core.** The streaming 1D-FFT kernel that operates on the SRAM-buffered $k$-row working set is automatically generated using the public Spiral online tool [3]. The Spiral generator can produce fully pipelined 1D-FFT cores over a wide range of user-selected processing rates for a commensurate charge in logic cost. For a balanced design, we generate the 1D-FFT pipeline that exactly matches the data rate of our DRAM reading and writing bandwidth, so the 1D-FFT core performance is neither the bottleneck nor unnecessarily high.

## V. Results

In this section, we evaluate the performance, bandwidth efficiency, and power efficiency of our 2D-FFT implementations by comparing against both existing hardware (FPGA, ASIC) and software implementations (CPU, GPU). Our implementations are targeted to the Altera Stratix IV EP4SGX530 FPGA on the DE4 development board. Our implementations support complex double-precision floating point values (2×64 bits per complex word) which are required in realistic use scenarios for large 2D-FFTs to preserve sufficient accuracy in the final result (particularly in scientific computing applications).

### A. Performance Comparisons

We evaluate our implementations along three metrics: raw performance, bandwidth efficiency, and power efficiency. As is customary in the FFT literature, our raw performance metric is reported in "pseudo" billion floating point operations per second, (Gflop/s); this metric is calculated as $1/\text{runtime}$ scaled by a constant that is the standard nominal operation count[1]. We define bandwidth efficiency as performance normalized to available memory bandwidth, (Gflop/s)/(GByte/s), and power efficiency as performance normalized to power consumed, (Gflop/s)/Watt). Higher values are better for all metrics.

**Against other ASIC and FPGA implementations.** In Table I, we first benchmark our performance against other hardware solutions (ASIC and FPGA) found in the literature. Our implementation outperformed all benchmarked implementations except one design that used 16-bit fixed-point and omitted the non-trivial final bit-reversal permutation. An important caveat to keep in mind is that this comparison of published designs is based on self-reported performance achieved on different platforms, memory systems, and data precisions. To give an absolute sense of quality, however, the performance of our implementation is within 11% of an idealized platform with infinitely fast on-chip processing and a perfect off-chip memory system that has a bounded bandwidth but no access latency, row buffer miss penalty, or refresh penalty.

[1] $5n^2 \log_2 n^2$, for $n$-by-$n$ 2D-FFT

Table I
PERFORMANCE COMPARISON WITH EXISTING HARDWARE SOLUTIONS (1,024-BY-1,024 2D-FFT).

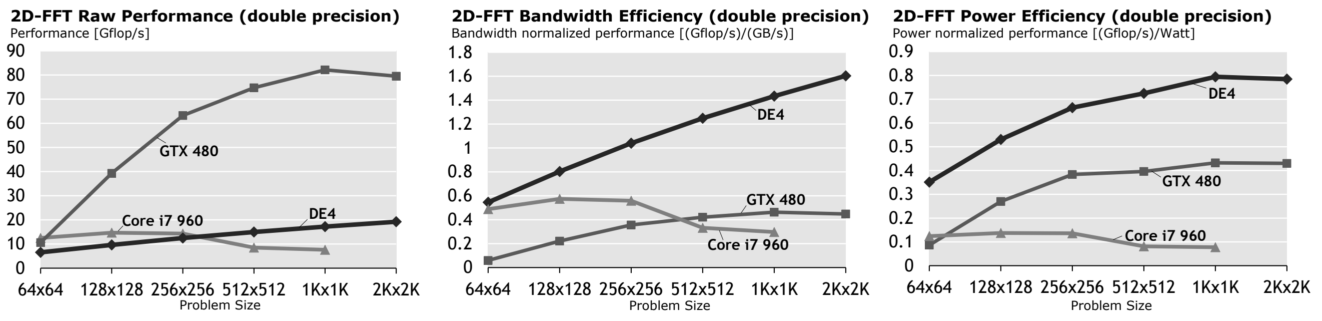
| Platform | Memory | Precision (bits) | Runtime (ms) | Source |
|---|---|---|---|---|
| Virtex-5 LX155 | 1xDDR2-400 | 32 (single) | 102.6 | [4] |
| Virtex-E | 4xSRAM | 16 (fixed) | 62.5 | [5] |
| ASIC (180nm) | 4xSDRAM | 32 (single) | 21 | [6] |
| Virtex-5 FX | 1xDDR2-400 | 16 (fixed) | $5.5^{(1)}$ | [7] |
| Stratix IV | 2xDDR2-800 | 64 (double) | 6.1 | Ours |

(1) In this implementation a final bit-reversal permutation is separated from 2D-FFT computation and handled by the host computer.

**Against CPUs and GPUs.** Figure 3(a) compares our raw performance to the best-available platform-tuned software solutions running on a quad-core 3.2 GHz Intel Core i7 960 CPU and an NVIDIA GeForce GTX 480 GPU. For the Core i7 we used the Spiral framework [8]; for the GTX 480 we used CUFFT 4.0 [9]. For this comparison, all platforms (summarized in Table 3) are running exactly the same application: double-precision 2D-FFT.

In terms of raw performance the GTX 480 dominates for almost all problem sizes. This is not too surprising since the GTX 480 enjoys 177.4 GByte/s of memory bandwidth. When the Core i7 and the DE4 are compared, we observe that for small problem sizes (e.g. $\leq 256 \times 256$), the Core i7 outperforms the DE4 implementation. Once the problem size exceeds the Core i7's 8 MByte L3 cache, its performance degrades to about half of our implementation's (even though the Core i7 has 25.6 GByte/s of memory bandwidth, more than twice the DE4's).

We next direct our attention to bandwidth and power efficiency. Going forward, off-chip bandwidth and power budgets will not grow as fast as on-chip processing performance. A comparison of bandwidth efficiency and power efficiency answers the question: for a given amount of memory bandwidth and power on a hypothetical future computing device, how do you extract the maximum performance? Figure 3(b) reports the comparison of memory bandwidth efficiency of the 2D-FFT running on the Altera DE4, Intel Core i7 and NVIDIA GTX 480. The DE4 implementations have significantly better bandwidth efficiency over all problem sizes relative to both GTX 480 and Core i7. For the problem size of 2,048-by-2,048, we could produce a re-tuned implementation that equals the absolute performance of the



(a) Raw performance (b) Bandwidth efficiency (c) Power efficiency

Figure 3. Raw performance, bandwidth efficiency, and power efficiency results for FPGA (DE4), CPU (Core i7 960) and GPU (GTX 480).

Table II
COMPARISON OF THE PLATFORMS.

| | Core i7 960 | GTX 480 | Stratix IV EP4SGX530 |
|---|---|---|---|
| DRAM Type | DDR3 | GDDR5 | DDR2 |
| # of Memory Channels | 3 | 6 | 2 |
| Off-chip BW (GByte/s) | 25.6 | 177.4 | 12 |
| On-chip Memory (MByte) | 8 | 1.69 | 2.53 |
| Proc. Frequency (MHz) | 3,200 | 1,401 | 200 |

GTX 480 if we had available an FPGA platform with 49.6 GByte/s (just 28% of the bandwidth available to the GTX 480). Figure 3(c) next reports the comparison of the power efficiency. All power numbers reported in this figure are measured on actual systems and include DRAM power. The DE4 implementations offer the best power efficiency, about twice the efficiency of the GTX 480, for all problem sizes.

## VI. RELATED WORK

There have been many implementations of 2D-FFT on a variety of platforms. These include software implementations on CPUs [8] and GPUs [9]. There are also examples of ASIC-based [6] and FPGA-based 2D-FFT implementations. Some FPGA examples only consider on-chip operation and assume the dataset fits in on-chip memory [10]. FPGA examples that consider off-chip memory interfacing include [7], [4], [5]. Among these implementations, few directly addressed the efficient utilization of the off-chip memory bandwidth. The designs in [7] and [4] do address the memory bandwidth problem but not at the level of detail that includes DRAM row buffer effects. None of the prior FPGA-based implementations targeted double-precision arithmetic which is the required norm for the problem sizes we are concerned with.

## VII. CONCLUSIONS

For 2D-FFTs on large data sets, the main performance bottleneck is the off-chip memory bandwidth. Traditional row-column or transpose-based algorithms do not exploit off-chip memory bandwidth efficiently due to poor memory access patterns. In this work, we addressed this issue through a joint optimization of algorithm and architecture. Our effort resulted in highly optimized 2D-FFT implementations based on an algorithm designed specifically for efficient use of the DRAM row buffer. Our implementations make efficient use of the on-chip compute resources and can sustain the maximum off-chip memory bandwidth throughout the 2D-FFT computation. Our results showed that in raw performance at the target problem sizes, we significantly outperform other comparable FPGA-based solutions. Our architecture is also more efficient than the best-available software running on state-of-the-art GPUs and CPUs in terms of the ratio between achieved performance and available off-chip bandwidth and the ratio between achieved performance and power consumption.

## ACKNOWLEDGEMENTS

The authors acknowledge the support of the C2S2 Focus Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. Funding for this work was also provided in part by NSF CCF-1012851 and Altera. The authors thank Altera for their FPGA and tool donations.

## REFERENCES

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

[2] M. Püschel *et al.*, "Permuting streaming data using RAMs," *Journal of the ACM*, vol. 56, no. 2, pp. 10:1–10:34, 2009.

[3] "Spiral DFT/FFT IP generator," http://www.spiral.net/hardware/dftgen.html.

[4] C.-L. Yu *et al.*, "Multidimensional DFT IP generator for FPGA platforms," *IEEE Transactions on Circuits and Systems*, vol. 58, no. 4, pp. 755–764, 2010.

[5] I. S. Uzun *et al.*, "FPGA implementations of fast Fourier transforms for real-time signal and image processing," in *IEEE Conference on Field-Programmable Technology (FPT)*, 2003, pp. 102–109.

[6] "PowerFFT ASIC," http://www.eonic.com/.

[7] C.-L. Yu *et al.*, "FPGA architecture for 2D discrete Fourier transform based on 2D decomposition for large-sized data," *Journal of Signal Processing Systems*, vol. 64, no. 1, pp. 109–122, 2011.

[8] "Spiral," http://spiral.net/.

[9] "CUDA 4.0 CUFFT," http://developer.nvidia.com/cuFFT.

[10] P. A. Milder *et al.*, "Formal datapath representation and manipulation for implementing DSP transforms," in *Design Automation Conference (DAC)*, 2008, pp. 385–390.