Journal of Signal Processing and Systems manuscript No.
(will be inserted by the editor)

A Highly Efficient Multicore Floating-Point FFT Architecture Based on Hybrid Linear Algebra/FFT Cores

Ardavan Pedram · John D. McCalpin · Andreas Gerstlauer

Received: date / Accepted: date

Abstract FFT algorithms have memory access patterns that prevent many architectures from achieving high computational utilization, particularly when parallel processing is required to achieve the desired levels of performance. Starting with a highly efficient hybrid linear algebra/FFT core, we co-design the on-chip memory hierarchy, on-chip interconnect, and FFT algorithms for a multicore FFT processor. We show that it is possible to to achieve excellent parallel scaling while maintaining power and area efficiency comparable to that of the single-core solution. The result is an architecture that can effectively use up to 16 hybrid cores for transform sizes that can be contained in on-chip SRAM. When configured with 12MiB of on-chip SRAM, our technology evaluation shows that the proposed 16-core FFT accelerator should sustain 388 GFLOPS of nominal double-precision performance, with power and area efficiencies of 30 GFLOPS/W and 2.66 GFLOPS/mm^2, respectively.

This research was partially sponsored by NSF grants CCF-1218483 (Pedram and Gerstlauer), CCF-1240652 and ACI-1134872 (McCalpin) and also NASA grant NNX08AD58G (Pedram). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF) or the National Aeronautics and Space Administration (NASA).

Ardavan Pedram and Andreas Gerstlauer
Department of Electrical and Computer Engineering
The University of Texas at Austin
E-mail: {ardavan,gerstl}@utexas.edu

John D. McCalpin
Texas Advanced Computing Center
The University of Texas at Austin
E-mail: mccalpin@tacc.utexas.edu

Keywords Fast Fourier Transform · Floating Point · Multicore · memory hierarchy · power efficiency · low power design · linear algebra

1 Introduction

Both linear algebra (LA) and Fast Fourier Transforms (FFTs) are fundamentally linked to the underlying mathematics of many areas of computational science. Matrix computations form the basis of almost all numerical methods. By contrast, FFTs are perhaps the most important single tool in signal processing and analysis, and play a fundamental role in indirect imaging technologies, such as synthetic aperture radar [5] and computerized tomographic imaging [14]. FFTs are a widely used tool for the fast solution of partial differential equations, and support fast algorithms for the multiplication of very large integers. Fourier transforms and dense linear algebra operations are closely related, and many scientific applications (such as finite element or N-body problems) often require both. However, compared to the direct computation of the discrete Fourier transform by matrix computations, the FFT has a more modest number of computations per data element (this is one of the main reasons that it is "fast"), so that performance of FFT algorithms is typically limited by the data motion requirements rather than by the arithmetic computations.

Compared to dense linear algebra operations, FFTs provide many additional challenges to obtaining high performance. First: the increased ratio of data movement per computation (even with perfect caches) will cause the algorithm to be memory bandwidth limited on most current computer systems. Second: memory access patterns include strides of $2, 4, 8, ...N/2$, which

interfere pathologically with the cache indexing and the cache and memory banking for standard processor designs. Third: the butterfly operation contains more additions than multiplications, such that the balanced FPUs on most current architectures will be under-utilized.

In previous work, we analyzed the similarities between General Matrix-Matrix Multiply (GEMM) and FFT algorithms, and we showed how one might transform an optimized GEMM core into an FFT accelerator [26]. Our starting point was a Linear Algebra Core (LAC) [21] that can support the full range of level-3 BLAS [25] and matrix-factorizations [24] with high efficiency. After evaluating LAC limitations and tradeoffs for possible FFT solutions, we introduced a flexible, hybrid design that can perform both linear algebra and FFT computations efficiently. This hybrid core is based on a minimal set of modifications to the existing LAC architecture and is optimized for FFTs over a wide range of vector lengths.

In this paper, we continue our analysis of larger FFT problem sizes that cannot fit into the core-local memory, but are small enough to fit into an on-chip scratch-pad storage. We explore different options to accelerate such problems when exploiting multiple hybrid LA/FFT cores. In the process, we examine different possible modes of operation in which the design can run FFTs as a multi- or many-core solution for different problem sizes. We present the design of a unique low-power memory hierarchy architecture that provides sufficient bandwidth for the cores to perform FFTs efficiently. Finally, we study the power efficiency tradeoffs as function of the required memory bandwidth and storage size for different configurations.

The goal of this work is to explore the design space and perform a study to evaluate the potential for scalability of a multicore configuration of the hybrid linear algebra/FFT core of our previous work, rather than to advance a specific implementation thereof. We focus primarily on double-precision floating-point performance – in part because the original linear algebra core was designed for double precision, and in part because the use of double precision variables increases bandwidth requirements and thus increases the need to focus on optimizing the data motion in the architecture. We include some single precision comparisons to FPGAs and specialized FFT engines in the final section because we were unable to find comparable double-precision results in the literature.

In doing so, we evaluate and explore tradeoffs using analytical performance and power models that have been calibrated through synthesis of key components and comparisons against literature. Our methodology is based on multiple iterations of an algorithm-architecture co-design process, taking into account the interplay between design choices for the core and the on-chip memory hierarchy. As part of this process, we study multiple choices in multi-core configurations and compare them in terms of power, area efficiency, and design simplicity.

The rest of the paper is organized as follows: In Section 2 we provide a brief overview of related work. Section 3 describes the conventional and Fused Multiply Add (FMA) Radix-2 and Radix-4 butterfly FFT algorithms. In Section 4, we describe the baseline architecture of existing hybrid LAC/FFT cores. Section 5 then describes the mapping of larger FFTs onto a larger design, and Section 6 studies the trade-off and derivation of different architecture constraints. In Section 7 we present a set of architecture configurations for both single- and multi- core solution suggested by our analytical models. In Section 8, we present estimated performance and efficiency of our design and compare with some competing options. Finally, we conclude the paper in Section 9.

2 Related Work

The literature related to fixed-point FFT hardware in the digital signal processing domain is immense. Literature reviews of hardware implementations date back to 1969 [3] – only four years after the publication of the foundational Cooley-Tukey algorithm [7].

The literature related to specialized floating-point FFT hardware is considerably less broad, especially when excluding work that is concerned exclusively with FPGA implementations [1,11,17], or when seeking references to double-precision floating-point FFTs. Much of the literature focuses on the requirements of the arithmetic units (e.g., [35,30]), or on the construction of individual butterfly units [12,31], or on the performance for streams of fixed-sized FFT's (e.g., [20]) rather than on complete FFT algorithms for a variety of problem sizes (as is our focus here).

Important recent work concerns the automatic generation of optimized hardware FFT designs from high-level specifications [19]. These hardware designs can use either fixed or floating-point arithmetic and can be used in either ASIC or FPGA implementations. One double-precision result using FPGAs [1] explores the issue of obtaining maximum performance for large 2D FFTs from a fixed level of off-chip memory bandwidth, achieving greater than 85% of the theoretical performance limit for the off-chip bandwidth available. Hemmert and Underwood [11] provide performance comparisons between CPU and FPGA implementations of double-precision FFTs, and include projections of anticipated performance.

Performance of FFT algorithms varies dramatically across hardware platforms and software implementations, depending largely on the effort expended on optimizing data motion. The predominance of power-of-two memory access strides causes general purpose microprocessor based systems to deliver poor performance on FFTs that do not fit in cache, even with highly optimized implementations [8]. Programmable Graphics Processing Units (GPUs) often provide much higher performance for FFTs, but with only moderately higher energy and area efficiency than general-purpose processors [1,27]. Modifications to enhance FFT performance in the Godson-3B processor, a MIPS64-compatible, high-performance 8-core CPU, are presented in [18], and a highly optimized implementation of single precision FFT for the Cell BE is presented in [10]. A broad survey of the power, performance, and area characteristics of single-precision FFT performance on general-purpose processors, GPUs, FPGAs and ASICs is provided by Chung [6], while a more detailed comparison of single-precision FFT performance on recent FPGAs and GPUs is presented in [27].

3 FFT Core Algorithm

At the lowest level, FFT algorithms are based on combining a small number of complex input operands via sum, difference, and complex multiplications to produce an equal number of complex output operands. These are referred to as "butterfly" operations because of the shape of the dataflow diagram (e.g., as shown later in Fig. 3). In this section, we briefly give the mathematical description of Radix-2 and Radix-4 FFT butterfly operations as optimized for execution on Fused Multiply-Add (FMA) units. Then, we discuss the data communication patterns that are needed when applying these operations to compute FFTs of longer sequences.

The Radix-2 Butterfly operation can be written as the following matrix operation, where w_L^j are constant values (usually referred to as "twiddle factors") which we store in memory:

$$\begin{pmatrix} x(j) \\ x(j+L/2) \end{pmatrix} := \begin{pmatrix} 1 & \omega_L^j \\ 1 & -\omega_L^j \end{pmatrix} \begin{pmatrix} x(j) \\ x(j+L/2) \end{pmatrix}.$$

This operation contains a complex multiplication operation and two complex additions, corresponding to 10 real floating-point operations. Using a floating-point MAC unit, this operation takes six Multiply-ADD operations that yields 83% utilization.

A modified, FMA-optimized butterfly is introduced in [15], where the multiplier matrix in the normal butterfly is factored and replaced by:

$$\begin{pmatrix} 1 & \omega_L^j \\ 1 & -\omega_L^j \end{pmatrix} = \begin{pmatrix} 2 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & -\omega_L^j \end{pmatrix}.$$

This algorithm requires 12 floating point operations represented in six multiply-adds. Although the total number of floating-point operations is increased, they all utilize a fused multiply-add unit and the total number of FMAs remains six.

A Radix-4 FFT butterfly is typically represented as the following matrix operation:

$$\begin{pmatrix} x(j) \\ x(j+L/4) \\ x(j+L/2) \\ x(j+3L/4) \end{pmatrix} \times = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} \mathbf{diag}(1, \omega_L^j, \omega_L^{2j}, \omega_L^{3j}).$$

This contains three complex multiplications and eight complex additions that sum up to 34 real floating-point operations. The number of complex additions is much larger than the number of multiplications. Hence, there is a clear computation imbalance between multiplications and additions. Note also that three complex twiddle factors ω_L^j, $\omega_L^2 j$, and $\omega_L^3 j$ all have to be brought into the butterfly unit.

Alternately, the Radix-4 matrix above can be permuted and factored to give the following representation ($\omega = \omega_L^j$):

$$\begin{pmatrix} x(j) \\ x(j+L/4) \\ x(j+L/2) \\ x(j+3L/4) \end{pmatrix} \times = \left(\begin{array}{cc|cc} 1 & 0 & \omega & 1 \\ 0 & 1 & 0 & -i\omega \\ \hline 1 & 0 & -\omega & 0 \\ 0 & 1 & 0 & -i\omega \end{array}\right) \left(\begin{array}{cc|cc} 1 & \omega^2 & 0 & 0 \\ 1 & -\omega^2 & 0 & 0 \\ \hline 0 & 0 & 1 & \omega^2 \\ 0 & 0 & 1 & -\omega^2 \end{array}\right).$$

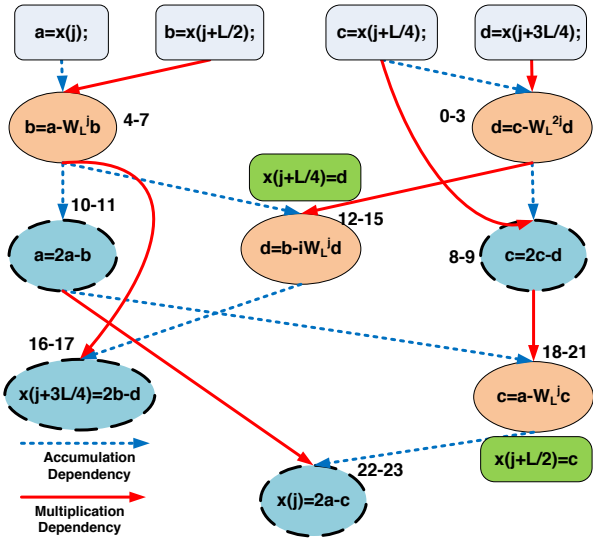

Fig. 1 DAG of the optimized Radix-4 butterfly using a fused multiply-add unit. Rectangles on top indicate the input data, solid nodes show complex computations with four FMA operations each, nodes with dashed lines show complex computations with two FMA operations each. The nodes are executed in an order that avoids data dependency hazards due to pipeline latencies, as shown by the start-finish cycle numbers next to each node.

This can be further divided recursively using the same factorization as in the radix-2 FMA-adapted version. The result generates 24 FMA operations as depicted in Fig. 1. The FMAC utilization for the Radix-4 DAG is 34/48=70.83%, but this corresponds to 40/48=83.33% if using the nominal $5NLog_2^N$ operation count from the Radix-2 algorithm that is traditionally used in computing the FLOP rate. Further details about this algorithm will be presented in Section 5. The number of loads also drops because only two of the three twiddle factors (ω_L^j and ω_L^{2j}) are required to perform the computations.

The two implementations of an L-point Radix-4 FFT are shown below. The pseudo-code for the standard implementation is shown on the left and the pseudo-code for the FMA optimized version is shown on the right:

for $j = 0 : L/4 - 1$	for $j = 0 : L/4 - 1$
$a := x(j);$	$a := x(j);$
$b := \omega_L^j x(j + L/4)$	$b := x(j + L/4)$
$c := \omega_L^{2j} x(j + L/2)$	$c := x(j + L/2)$
$d := \omega_L^{3j} x(j + 3L/4)$	$d := x(j + 3L/4)$
$\tau_0 := a + c$	$b := a - \omega_L^{2j} b$
$\tau_1 := a - c$	$a := 2a - b$
$\tau_2 := b + d$	$d := c - \omega_L^{2j} d$
$\tau_3 := b - d$	$c := 2c - d$
$x(j) := \tau_0 + \tau_2;$	$x(j + L/2) := c = a - \omega_L^j c$
$x(j + L/4) := \tau_1 - i\tau_3;$	$x(j) := 2a - c$
$x(j + L/2) := \tau_0 - \tau_2;$	$x(j + L/4) := d := b - i\omega_L^j d$
$x(j + 3L/4) := \tau_1 + i\tau_3;$	$x(j + 3L/4) := 2b - d$
end for	end for

4 Hybrid Core Design

The microarchitecture of our baseline hybrid Linear Algebra / FFT Core (LAFC) is illustrated in Fig. 2 [26]. The LAFC architecture consists of a 2D array of $n_r \times n_r$ Processing Elements (PEs), with $n_r = 4$ in Fig. 2. Each PE has a double-precision Floating-Point Multiply ACcumulate (FPMAC) unit with a local accumulator, and local memory (SRAM). PEs on the same row/column are connected by low-overhead horizontal/vertical broadcast buses. The control is distributed and each PE has a state machine that drives a predetermined, hard coded sequence of communication, storage, and computation steps for each supported operation.

4.1 Architecture

For performing linear algebra operations, the architecture and implementation optimize the rank-1 update operation that is the innermost kernel of parallel matrix multiplication [34]. This allows the implementation to achieve orders of magnitude better efficiency in power and area consumption than conventional general purpose architectures [23].

The FPMAC units perform the inner dot-product computations central to almost all linear algebra operations. To achieve high performance and register-level locality, the LAFC utilizes pipelined FPMAC units that can achieve a throughput of one *dependent* FPMAC operation per cycle [13], which avoids data dependency hazards and associated stalls in regular architectures [22].

For FFT operation, the off-core bandwidth needs to be double that of a pure linear algebra design. The PEs must be able to overlap the prefetching of input data and the post-storing of output data from/to off-core memory concurrently with the computations. The LAFC therefore has two external memory interfaces that allow both row and column buses to transfer data to/from PEs. This design is symmetric and natively supports transposition.

The PE-internal storage also needs enough bandwidth to provide data for both Radix-4 computations and off-core communications. The original linear algebra PE has one larger, single-ported and one smaller, dual-ported SRAM to save power for more frequent accesses to elements of Matrix B. Since the smaller SRAM is already dual ported, the larger SRAM is further divided into two halves for FFT operation. Furthermore, the register file is extended with more ports and more capacity to match the requirements of the FFT. An 8-word register file is needed to store the four complex input, temporary, and output values of the FMA-optimized Radix-4 butterfly. Overall, the datapath within the LAFC has a symmetric design with two separate buses each connected to all the components in the PE and to one of the SRAMs.

4.2 FFT Mapping

The broadcast bus topology allows a PE to communicate with other PEs in the same row and with other PEs in the same column simultaneously. To maximize locality, we consider only designs in which each butterfly operation is computed by a single PE, with communication taking place between the butterfly computational steps. We note that if the LAFC dimensions are selected as powers of two, the communication across PEs between both Radix-2 or Radix-4 butterfly operations will be limited to the neighbors on the same row or column. The choice of $n_r = 2$ provides little parallelism, while values of $n_r >= 8$ provide inadequate bandwidth per PE due to the shared bus interconnect. Therefore, we choose $n_r = 4$ as the standard configuration for the rest of the paper.

A Radix-2 operation takes six FMA operations. Performing Radix-2 operations in each PE, the LAFC can perform 32-point FFTs, but can only hide the latency

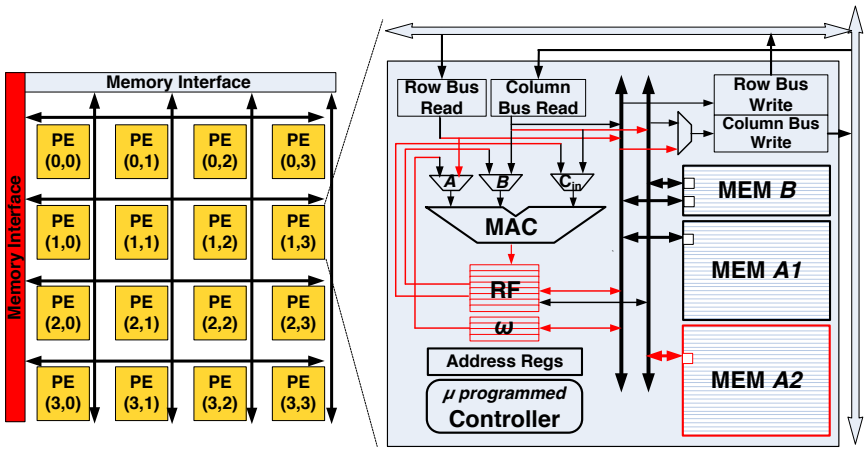

Fig. 2 Hybrid linear algebra/FFT core (LAFC) with support for reuse of temporary values of Radix-4 butterfly operations and enough bandwidth between PEs and to off-core memory. Extensions for FFT operation are shown in red.

of the FMA pipeline for FFT transforms with 64 or more points. The shortcoming of performing Radix-2 butterflies on the PEs comes from a computation/ communication imbalance [26]. In stages two through four of the overall algorithm, broadcast buses are being used for exchanging data. In doing so, n_r complex numbers are transferred on the bus, which takes $2n_r$ (eight) cycles. Since computation requires only six cycles, this imbalance decreases utilization by an undesirable 25%.

The Radix-4 butterfly on the PE is more complicated. Fig. 1 shows the DAG, where solid ellipse nodes take 4 FMA operations and dashed nodes take 2 FMA operations. A pipelined FPMAC unit has q pipeline stages with $q = 5 \sim 9$. The nodes in the DAG should be scheduled in a way that data dependency hazards do not occur due to pipeline latency. However, the FP-MAC units have single cycle accumulation capabilities. Hence, no data dependency hazards can occur among addition/accumulations (dashed arrows). For the multiplication dependencies (solid arrows), there should be at least q cycles between start of a child node and the last cycle of its parent. The start-finish cycle numbers next to each node show an execution schedule that tolerates pipeline latencies of up to 9 cycles with no stalls, thus providing 100% FMA utilization.

The overall Radix-4 algorithm is similar to the Radix-2 algorithm, but with more work done per step and with communication performed over larger distances in each step. Fig. 3 shows a 64-point FFT where each PE performs Radix-4 butterfly operations. This transform contains three stages. The communication pattern for the first PE in the second and third stages is shown with highlighted lines in the figure. In the second stage, PE_0=PE(0,0) has to exchange its last three outputs with the first outputs of its three neighboring PEs:

$$\text{PEs}_{(1,2,3)\times 4^0} \Rightarrow \begin{array}{r} \text{PE}_1 = \text{PE}(0,1) \\ \text{PE}_2 = \text{PE}(0,2) \\ \text{PE}_3 = \text{PE}(0,3) \end{array}$$

(See Fig. 4). Similarly, in the third stage, PE(0,0) has to exchange its last three outputs with the first outputs of PEs that have distance with multiples of 4:

$$\text{PEs}_{(4,8,12)} = \text{PE}_{(1,2,3)\times 4^1} \Rightarrow \begin{array}{r} \text{PE}_4 = \text{PE}(1,0) \\ \text{PE}_8 = \text{PE}(2,0) \\ \text{PE}_{12} = \text{PE}(3,0) \end{array}$$

Since there are only 16 PEs in a core, PEs that have distances of multiples of $4^2 = 16$ fall onto the same PE, and there is no PE-to-PE communication. When communication is required, all the PEs on the same row or column have to send and receive a complex number to/from each of their neighbors. The amount of data that needs to be transferred between PEs is $2n_r(n_r-1)$. For the case of $n_r = 4$, the communication takes 24 cycles, which exactly matches the required cycle count for the radix-4 computations. As such, the remainder of the paper will focus on the Radix-4 solution only.

The approach used for the 64-point FFT can be generalized to any (power of 4) size for which the data and twiddle factors fit into the local memory of the PEs. Consider an $N = 4^m$ point FFT using the Radix-4 butterfly implementation described above. The transform includes $log_4^N = m$ stages. Out of these m stages, only two use broadcast buses for data transfer – one stage using the row buses and one stage using the column buses. The rest of data reordering is done by address replacement locally in each PE. Therefore as discussed in the next section, as the transform size increases, the broadcast buses are available for bringing data in and out of the LAC for an increasing percentage of the total time.

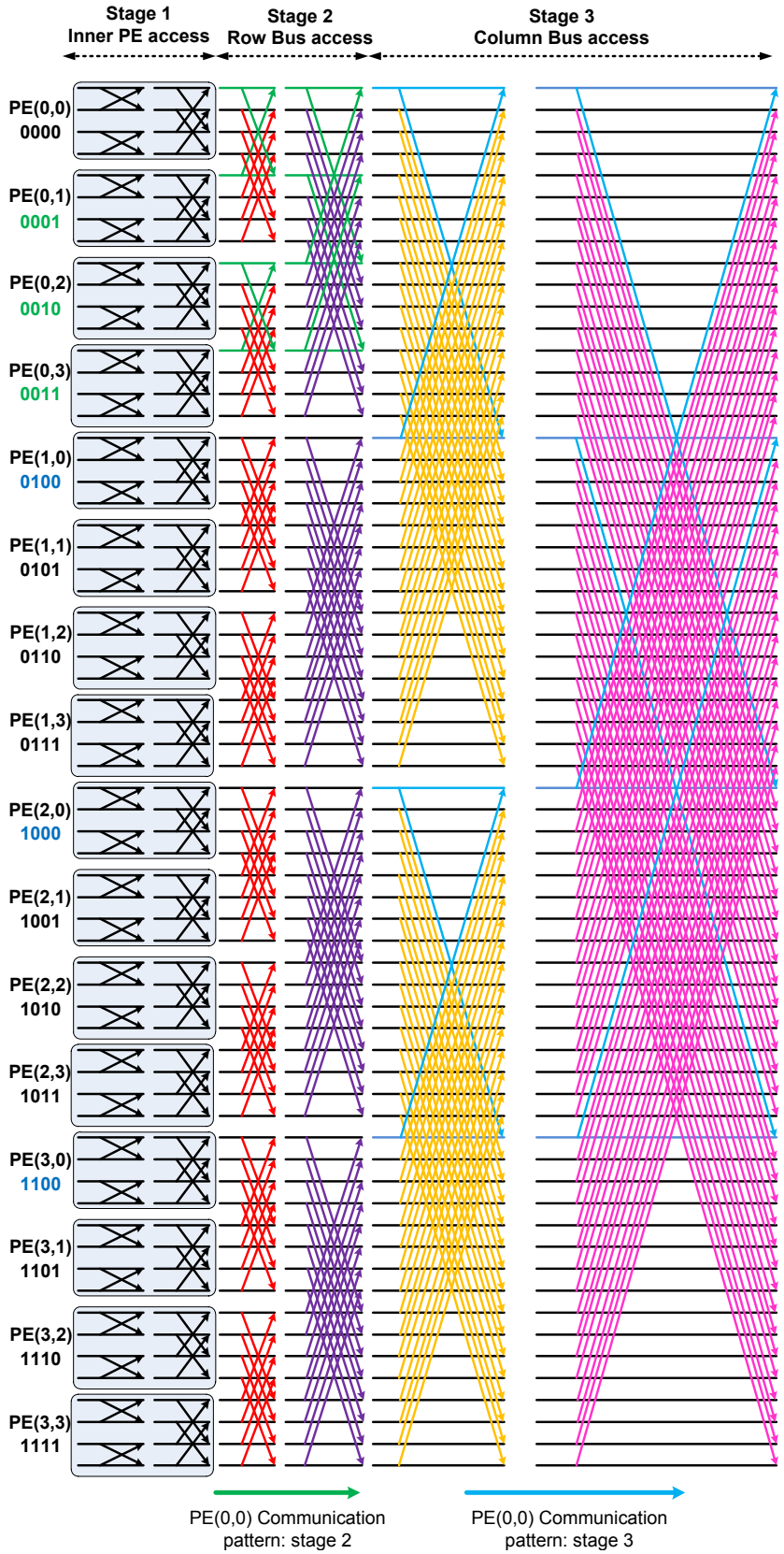

Fig. 3 64 point FFT performed by 16 PEs in the core. Each PE is performing Radix-4 Butterfly operations. The access patterns for PE(0,0) are highlighted. Stage 2 only utilizes row-buses to perform data communications. Stage 3 only utilizes column-buses to perform data communications.

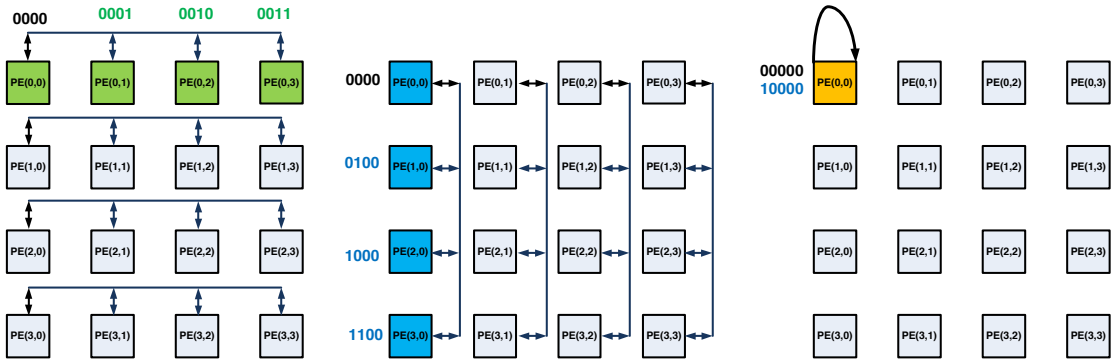

Fig. 4 Data communication access patterns between PEs of the LAFC for Radix-4 FFT.

For larger problem sizes, the radix computations can be performed in a depth-first or breadth-first order (or in some combination). We choose the breadth-first approach due to its greater symmetry and simpler control. In this approach, all the butterflies for each stage are performed before beginning the butterflies for the next stage.

5 Mapping for Larger Transforms

The local memory in the PEs will allow storage of input data, output data, and twiddle factors for problems significantly larger than the 64-element example above, but the local memory size will still be limited. To make the examples concrete, we will use 4096 as a "typical" value for the maximum size that can be transformed in core-local memory, with more discussion in later sections.

Given a core capable of computing FFTs for vectors of length $[64, \ldots, 4096]$, it is of interest to explore the off-core memory requirements to support the data access patterns required by these small FFTs as well as those of more general transforms, such as larger 1D FFTs or multidimensional FFTs.

First, we note that the butterfly computations shown in Fig. 3 produce results in bit-reversed order. Although some algorithms are capable of working with transformed results in permuted orders, in general it is necessary to invert this permutation to restore the results to their natural order. Converting from bit-reversed to natural order (or the converse) generates many power-of-two address strides, which are problematic for memory systems based on power-of-two banking with multi-cycle bank cycle times. The most straightforward solutions are based on high-speed SRAM arrays capable of sourcing or sinking contiguous, strided, or random addresses at a rate matching or exceeding the bandwidth requirement of the core. These will be referred to as "off-core SRAM" in the discussions below.

In part due to this requirement for high-speed random access, this analysis is limited to sizes that are appropriate for on-chip (but off-core) memory. Considerations for off-chip memory are beyond the scope of this paper and are deferred to future work.

5.1 Mapping for Larger 1D FFTs

Support for larger one-dimensional FFTs is provided through the generalized Cooley-Tukey factorization, commonly referred to as the "four-step" algorithm [2]. For an FFT of length N, we split the length into the product of two integer factors, $N = N_1 \times N_2$ and treat the data as a two-dimensional array with N_1 columns and N_2 rows. The 1D discrete Fourier transform can then be computed by the sequence: (1) Transform the "columns" (N_1 FFTs of length N_2); (2) Multiply the results by an array of complex roots of unity (also referred to as "twiddle factors" in the literature, but these are an additional set of scaling factors beyond those included in the row and column transforms themselves[1]); (3) Transform the "rows" (N_2 FFTs of length N_1). The fourth step (transposing the output of step three) is omitted here, as discussed below.

For a core capable of directly performing transforms of up 4096 elements, this algorithm allows computing a 1D transform for lengths of up to $4096^2 = 2^{24} \simeq 16$ million elements. (On-chip memory capacity will not be adequate for the largest sizes, but the algorithm suffices for this full range.) For transforms larger than $N = 2^{24}$ elements, the transform must be factored more than two sub-factors, each less than or equal to 2^{12}. The corresponding algorithms can be derived directly or recursively, but support for these algorithms is beyond the scope of this paper.

[1] To avoid confusion in this paper, we distinguish between "local twiddle factors" used in the radix-4 operator and "global twiddle factors" used in the four-step method.

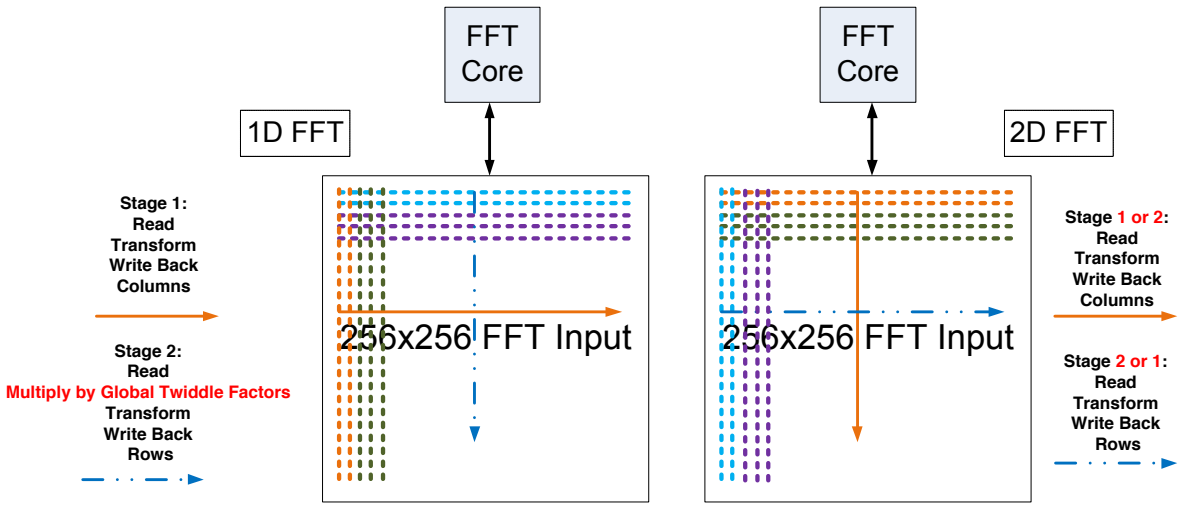

Fig. 5 Overview of data motion to/from the core for performing a 64K 1D FFT (left), and for a 256×256 2D FFT (right). Differences between the two cases are highlighted in red text.

The overall data motion for the 1D FFT is shown in in the left panel of Fig. 5. Assuming that the data begins in natural order, the first set of transforms must operate on strided data – essentially the "columns" of a row-major array. In order to perform transforms on naturally ordered data, an array transposition is required. This is performed while the data is being loaded from the on-chip memory into the core, and requires support for reading data at a stride of N_2 complex elements. The core performs the transform as described in the previous section. Each "column" of results is written back to the same location in off-core SRAM from which it was initially fetched. Note that the data in the core is in bit-reversed order after being transformed, so in order to return the data to off-core SRAM in natural order, the data must be loaded from the core in bit-reversed order, then returned to the off-core SRAM with the stride of N_2 complex elements.

The multiplication by the "global twiddle factors" can be performed at the end of the first set of transforms, or as a separate step, or (as in this design), as the data is read back into the core for the second set of transforms, but before the second set of transforms begins. The second FFT step therefore requires twice the read bandwidth of the first set of FFTs, since both the intermediate data and the global twiddle factors must be read from the off-core SRAM to the core for this step.

After the intermediate data is read into the core and multiplied by the global twiddle factors, the second set of transforms can be performed on each "row". As each of these row transforms is completed, the data is written from core memory to the off-core SRAM with the bit-reversal permutation to return the data to natural order.

This completes the computations for the 1D FFT, but at this point the values are stored in the transpose of the natural order. Given the ability of the off-core SRAM to source data in arbitrary order, it is assumed that subsequent computational steps will simply load the data in transposed order. (Since transforms can be factored in more than one way, this requires that subsequent steps know which factorization was used.)

5.2 Algorithm for Multidimensional FFTs

For a core capable of computing 1D FFTs of lengths $64, \ldots, 4096$, multi-dimensional FFTs of sizes up to 4096 in any dimension are straightforward, as they are simply the result of independent 1D transforms for each "row", "column", etc. The 2D transforms, in particular, are similar to large 1D FFTs computed by the four-step method, but are simpler to implement since no "global twiddle factors" are required, and since the final result does not require an additional transposition.

The data motion for the 2D FFT is shown in the right panel of Fig. 5. After the row transforms are produced and written back to the off-core SRAM (using bit-reversal to obtain natural ordering), the data is read in transposed order to allow computation of the column transforms. These are then written back in transposed order (with bit reversal to restore natural order) to complete the 2D FFT.

Multidimensional FFTs of higher order require a different stride for the transposition for each dimension, but are otherwise almost identical to the 2D FFT. For multidimensional transforms with any dimension(s)

larger than 4096, the four-step method described above for 1D transforms is applied to each transform of length greater than 4096.

5.3 Mapping of Transforms on Multiple Cores

The four-step algorithm for 1D FFTs described in Section 5.1 provides a clean decomposition of the FFT algorithm into independent "row" and "column" transforms, plus a point-wise set of global twiddle factor multiplications. This provides a large degree of computational parallelism that can be exploited across multiple cores in all three computational steps.

At a high level, the four-step algorithm used on multiple cores is almost the same as for the single-core case. For the first step, instead of fetching one column from off-core SRAM and processing it with one core, multiple columns are fetched, each column is sent to a core, and the transforms proceed concurrently. The global twiddle factor multiplication is point-wise, and so trivially parallelizable. Finally, multiple rows are fetched, each row is sent to a core, and again the transforms are performed concurrently.

To view the algorithm at a finer level of detail one must make assumptions about the memory hierarchy and interconnection topology.

For multidimensional transforms on multiple cores, the considerations of Section 5.2 apply. Instead of processing the independent transforms sequentially through a single core, they are block-distributed and processed sequentially by multiple cores. The off-core memory must be capable of handling different strides for each of the (non-unit-stride) dimensions.

5.4 Mappings and Their Modes of Operation

The modes in which an FFT engine might operate depend on the relative values of key attributes of the implementation technology available. Based on the preceding analysis, we can consider three cases:

1. *Core-contained*: Short transforms (64 to 4096 elements) that can be performed fully within the local memory of a core. These are implemented directly by repeated application of the Radix-4 kernel.
2. *Four-step, Single-Core*: Larger transforms executed by one core that must stage the data to/from an off-core (but on-chip) memory, using the four-step algorithm presented in Section 5.1.
3. *Multicore*: Transforms that are distributed across multiple cores to obtain better performance on each transform.

$\log_2 N$	MiB/array	1 Core	4 Core	16 Core
12	0.0625	direct	4-step*	4-step*
14	0.25	4-step	4-step*	4-step*
16	1.0	4-step	4-step	4-step*
18	4.0	4-step	4-step	4-step

Table 1 Transform sizes and algorithms for 1, 4, and 16 cores. "Direct": performed directly using radix-4 operations in the core's local memory. "4-step": uses the 4-step algorithm with global twiddle factors loaded from off-core SRAM. "4-step*": uses the 4-step algorithm, with global twiddle factors pre-loaded into core-local memory.

Case 1 (core-contained transforms) often occur as streams of independent vectors. These are trivial to pipeline (given adequate buffer space in the core and adequate off-core bandwidth) and trivial to parallelize (across independent transforms) and so will not be further discussed as an independent workload. However, these short core-contained transforms form the basis of the four-step method. Therefore, this mode of operation needs to be considered as a component of larger transforms

Case 2 (the four-step method on a single core) was initially presented in our previous work [26]. This mode forms the basis for the multicore algorithm that is our primary focus in this paper.

Case 3 (transform using multiple cores) is the main focus of the current work. It is closely related to Case 1 and Case 2, but introduces new balances of compute versus bandwidth and requires the introduction of an architecture for global communication across cores to support the transpose functionality.

In summary, the transform sizes being analyzed here and the corresponding choices of algorithms are presented in Table 1. The notation "4-step*" is used to refer to those cases for which the local memory requirements for working storage and buffers is small enough to allow the global twiddle factors to be pre-loaded into core-local memory.

6 Architecture Trade-off Analysis

In previous sections, we provided the fundamentals for mapping a Radix-4 based FFT transform onto a LAFC-based architecture. In this section, we derive per-core analytical models for architectural constraints and trade-offs in relation to various operation modes and architectural configuration parameters. Note that the tightest limitations always come from the 4-step method, so we will primarily focus on this mode.

We briefly review the primary core constraints and trade-offs for local storage, computation/communication overlap, and off-core bandwidth that are thoroughly

FFT $N_2 \times N_1$	2D No-Ov	2D Ov	1D No-Ov	1D Ov
Local Store	$4N_1$	$6N_1$	$6N_1$	$8N_1$
Radix-4 Cycl.	$6N_1 log_4 N_1/n_r^2$			
Tw. Mult. Cycl.	-	-	$6N_1/n_r^2$	$4N_1/n_r^2$
Comm	$2N_1$(R)+$2N_1$(W)		$4N_1$(R)+$2N_1$(W)	

Table 2 Single core storage and compute requirements for overlapped and non-overlapped versions of $N_2 \times N_1$ 2D and $N = N_2 \times N_1$ 1D FFTs for 4-Step mode.

discussed in [26]. The number of PEs in each row/column is denoted with n_r(=4) and problem sizes are chosen in the range of $N = 64, \ldots, 4096$. Each FMA-optimized Radix-4 butterfly takes 24 cycles as presented in Section 3. Therefore, an N-point FFT requires a cycle count of $N/4 \times 24 \times log_4^N/n_r^2$.

Similar to our linear algebra design [21], we consider two cases in our analysis for FFT on the core: no or full overlap of off-core communication with computation. Note that due to the high ratio of communication to computation ($O(N)/O(N \log N)$), the non-overlapped FFT solution suffers significantly resulting in low utilization. Hence, our primary focus will be on implementations that overlap computation and off-core data motion as completely as possible. The core requirements for these combinations are summarized in Table 2.

6.1 Single-core constraints for 2D FFTs

For both stages of the 2D FFT and the first stage of the 1D FFT, each core is performing independent FFT operations on rows or columns. The local twiddle factors remain the same for each transform, and can therefore be retained in core-local memory. Therefore, the core bandwidth and local store size can be calculated as follows: The amount of data to transfer for a problem of size N includes N complex inputs and N complex transform outputs resulting in a total of $4N$ real number transfers. In case of no overlap, data transfer and computation are performed sequentially. For the case of full overlap, the average bandwidth for a FFT of size N can be derived from the division of the total data transfers by the total computation cycles as $\text{BW}_{Avg} = 2n_r^2/3\log_4^N$. However, out of log_4^N stages, stage 2 utilizes row buses and stage 3 uses column buses for inter-PE communications. If column buses are used to bring data in and out of the PEs, the effective required bandwidth is increased to $\text{BW}_{eff} = 2n_r^2/3(\log_4^N - 1)$.

The aggregate local store of PEs includes the N complex input points and N complex local twiddle factors. In the non-overlapped case, this amount of storage suffices. There is no need for extra buffering capacity. However, the overlapped case requires an extra N point buffer to hold the prefetched input values for the next transform. Therefore, the aggregate PE local stores in a core should be $6N$ double-precision floating-point values.

6.2 Single-core constraints for 1D FFTs

In order to maximize the potential for overlapping computation with data transfer in the four-step method, the multiplication of intermediate results by the global twiddle factors is merged with the second set of transforms. In this merged step, the core must read both the intermediate row data and the global twiddle factors, then multiply them (point-wise) before performing the row transforms. The complex multiplication for each element adds $4N$ real multiplications to the total computations of this step, giving a total cycle count of $(6Nlog_4^N + 4N)/n_r^2$. The amount of data to transfer for a problem of size N includes $2N$ complex inputs (transform inputs and global twiddle factors), and N complex outputs resulting in a total of $6N$ real number transfers. In case of no overlap, data transfer and computation are performed sequentially. For the case of full overlap, the average bandwidth for an FFT of size N therefore becomes $\text{BW}_{Avg} = 3n_r^2/(3\log_4^N + 2)$. If column buses are used to bring data in and out of the PEs, the effective required bandwidth is increased (as in the 2D case described above) to $\text{BW}_{eff} = 3n_r^2/(3\log_4^N - 1)$.

Each N-point input to the core has to be multiplied by a different set of global twiddle factors, so another buffer is needed to pre-load the next column's global twiddle factors.

Finally, as described earlier, one can compute the 1D FFT by splitting N into the product of two integer factors, $N = N_1 \times N_2$. Earlier we noted that the fully-overlapped solution has lower communication load for larger transform lengths. Noting also that the second set of FFTs puts more communication load on the core/external memory, we expect that ordering the factors so that the larger factor corresponds to the length of the second set of transforms will provide more balanced memory transfer requirements. Fig. 7 demonstrates this effect for the case of a 64K point 1D FFT with three different options for $64K = N_1 \times N_2$.

6.3 Core constraints for multicore FFTs

For multicore operation, the constraints on the cores are similar or identical to those of the single-core case. However, the range of practical transform lengths is slightly modified by the use of the four-step algorithm (see Section 7.2). We therefore consider only sizes of 2^{12} or larger for a multicore solution. In this case, the

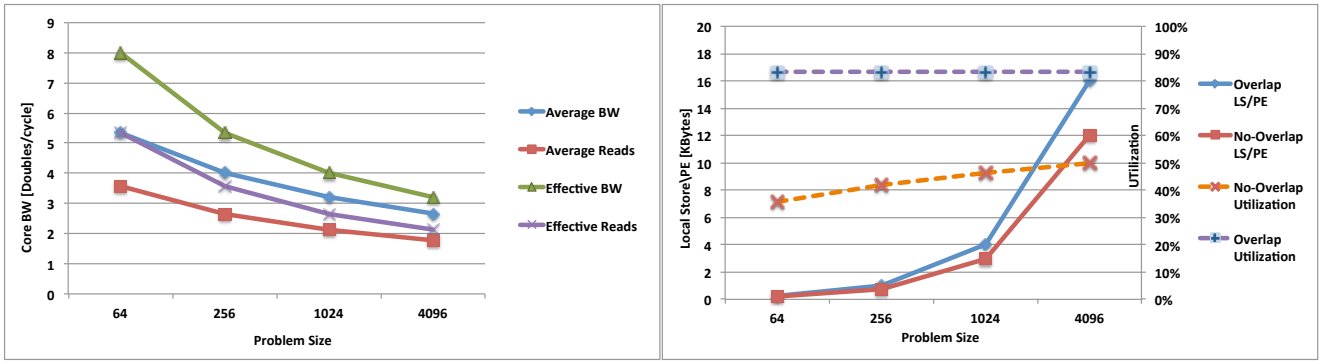

Fig. 6 Left: required bandwidth to support full overlap in the worst case for different problems. Note that eight doubles/cycle is the maximum capacity of a core with row and column buses used for external transfers. Right: local store per PE and respective utilization for both cases of non-overlapped and overlapped solutions.

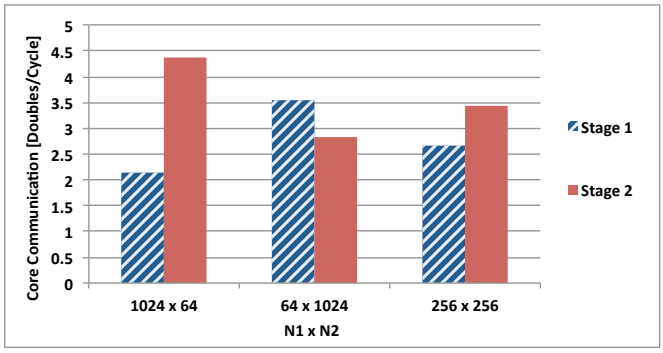

Fig. 7 Average communication load on core for the two stages of a 64K 1D FFT.

four-step method employs row and column transforms of length 2^6 or larger.

7 FFT Processor Architectures

In this section, we present a set of architecture configurations suggested by the preceding analyses. Several options for the shared memory hierarchy will be derived to meet the data handling and access pattern demands of the problem. We first start with the single-core architecture and then present the multi-core shared memory architecture.

7.1 Single Core Architecture

Fig. 6 shows the core bandwidth and local store requirements for the overlapped and non-overlapped algorithms. The utilization of the non-overlapped version increases from 35% to 50% as the size of the transform increases. The overlapped algorithm can fully utilize the FMAC units for all these sizes, maintaining its optimum utilization of slightly over 83.3%. Depending on the FFT type (1D or 2D), the overlapped algorithm requires 33%∼50% extra local storage. Note that the non-overlapped bandwidth is assumed to be at a fixed rate of four doubles/cycles, which is the maximum capacity of the LAFC with only one memory interface. For overlapped operation, we assume a full LAFC design with memory interfaces on both row and column busses.

Table 2 shows that the largest requirement for core-local storage is $8N_1$ words for a the second set of transforms in the four-step algorithm for the 1D FFT (i.e., four buffers of N_1 complex elements each). Noting that larger transform sizes require lower average off-core data transfer rates, there is a trade-off between core local storage size and off-core communication bandwidth. We have previously used the size $N = 4096$ as a baseline for largest core-containable transform size. Four buffers of this size requires 256 KiB of memory in the core, or 16 KiB per PE. Using the nomenclature of Figure 2, this corresponds to 8 KiB for each of MEM A1/A2 and 4 KiB for MEM B.

For the case of double-precision complex data, the natural data size is $2 \times 64 = 128$ bits, so we will assume 128-bit interfaces between the core buses and the off-core SRAMs. As shown in Section 5.1, the first step of a large 1D FFT requires less memory traffic than the second stage, which includes loading an additional set of twiddle factors. As such, we focus on the second stage here. We consider whether the instantaneous read and write requirements of the algorithm can be satisfied by two separate SRAMs, one for data and one for twiddle factors, each with a single 128-bit-wide port operating at twice the frequency of the core, giving each a bandwidth of 4 double-precision elements per cycle. (I.e., each SRAM has a bandwidth that exactly matches the bandwidth of the row buses or of column buses.)

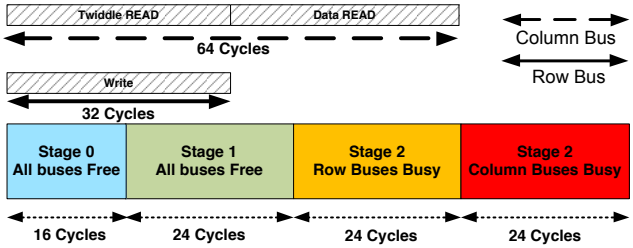

Fig. 8 Example scheduling of data bus usage for fully overlapped pre-fetch/post-store for the worst case considered – a 64-element transform in the second stage of the four-step algorithm. 75% of the free row and column bus cycles are required for off-core data transfers.

The worst case for this algorithm occurs for a second stage (row) transform length of $N_1 = 64$ elements, where full overlap of data transfers with computation requires that the external memory be able to provide 256 double-precision elements (64 complex input elements plus 64 complex twiddle factors) and receive 128 double-precision elements (64 complex output elements) during the 88 cycles required to perform the twiddle factor multiplications and the three Radix-4 computations. The proposed memory interface bandwidth is clearly adequate to provide for the average traffic – the SRAMs require 64 cycles (of the 88 available) to provide the 256 words prefetched by the core for its next iteration. The writes require only 32 of the 88 cycles, and these can be overlapped with the reads.

The core's row and column buses are not available for external data transfers in every cycle. Recall from Fig. 3 that the row buses are in use during the second Radix-4 step (24 cycles) of the 64-point FFT. By contrast, the column buses are in use during the third 24-cycle Radix-4 step. Both row and column buses are free during the twiddle factor multiplication step (16 cycles for $N_1 = 64$) and during the 24 cycles of the first Radix-4 computation. The detailed schedule can be implemented in a variety of ways, but the example in Fig. 8 shows that the required data transfers require only 75% of the row and column bus cycles that are available.

For all cases with $N > 64$, there are additional Radix-4 stages with no use of the row and column buses, making full overlap of communication and computation easier to schedule. As the transform size increases, the fraction of cycles available for data transfer increases slowly. When the four-step method is being used, the growth is slower than one might expect. The current design requires the bandwidth of two off-core SRAMs (i.e., one for twiddle factors and one for data input/output) for problem sizes up to 2^{16}, and the bandwidth of one off-core SRAM for larger sizes. Given the minimum of 12 MiB of SRAM required for the 2^{18} case, it seems likely that most larger problem sizes will have to be implemented using a different memory technology. Non-SRAM memories do not provide the low-latency, high-bandwidth, single-word random access capability assumed here, so larger problems will require algorithmic changes to accommodate the characteristics of the memory technology to be used.

7.2 Multicore Architecture

All FFT algorithms require some degree of global communication, either in the repeated application of radix butterflies of ever-increasing stride, or in the transposition operation of the four-step method. One option that might be considered to enable this global communication for a multicore architecture is the use of a single, shared multi-ported off-core SRAM. This approach, however, has serious scalability issues, with both area and energy per access growing up to quadratically with the number of ports, and with the maximum frequency of operation decreasing as the number of ports is increased. Therefore, a more elegant approach is required.

The treatment of the 1D vector as a 2D array in the four-step method suggests that distributing the data across the cores either by rows or by columns will enable half of the transforms to be performed entirely on local data. Since the "global twiddle factors" are only used in a point-wise fashion, they can always be distributed to allow local access, thus making approximately 3/5 of the accesses localizable.

These issues lead us to propose an architecture composed of a collection of "core plus private off-core SRAM memory", and assume that the data is block-distributed across the off-core SRAMs by row. (Due to the symmetry of the computations and data access, there are no fundamental differences between block distributions by row and by column.) With this assumption, the second set of transforms in the four-step method are on rows, and are therefore fully local in such an architecture. The first set of transforms, however, requires transposing the data across all of the cores. We therefore assume that the architecture contains a *transposer* that fully connects all the cores to all the off-core SRAMs, and that provides the transposition functionality that we obtained by simply using a strided access in the single-core case. This connectivity is illustrated schematically in Fig. 9 for the case of four cores. The operation of the transpose unit is illustrated schematically in Fig. 10 for the case of four cores. The ordering of the data accesses sent through the transpose unit is chosen to maximize the potential for overlap of computation with data transfers.

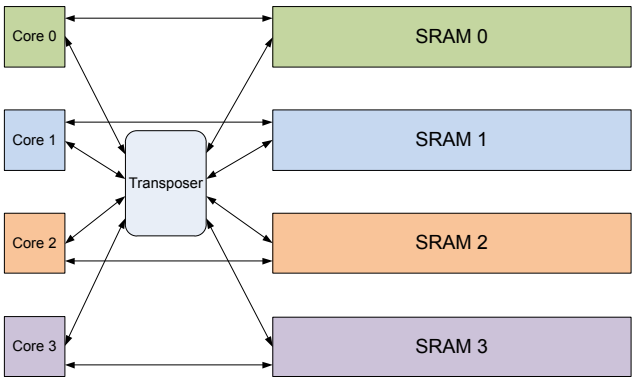

Fig. 9 Interconnections in a multicore system with four cores. The transpose unit interleaves the four input streams into the four output streams as shown in Fig. 10.

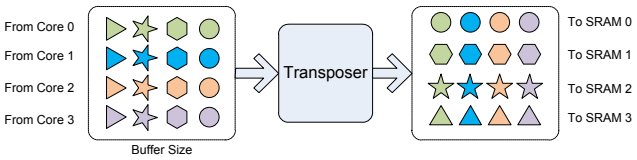

Fig. 10 Operation of transpose unit for a four-core system. The unit fetches four elements from each core (or SRAM), transposes the 4×4 array, then sends four elements to each SRAM (or core). This cycle is repeated until each core has received a full column, then repeated for each set of four columns in the 2D representation of the array.

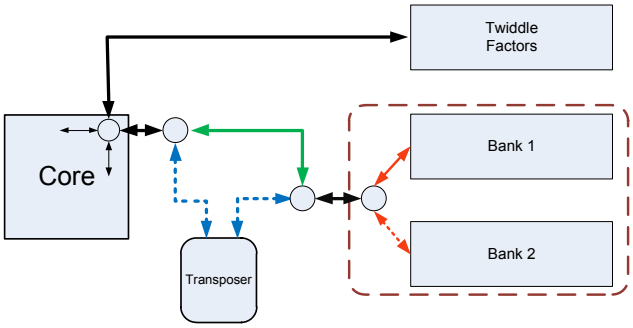

Fig. 11 Schematic of off core memory configuration in the multicore case. The transposer is connected to all cores and to all off-core data SRAM pairs.

We consider the memory configuration of a single core in more detail, as illustrated in Fig. 11. As in the single-core case, we assume that the data and twiddle factors are divided into separate SRAMs, each with a single 128-bit port running at twice the core frequency. We extend the configuration of [26] to support overlap of computation with off-chip data transfers by replicating the data SRAM. Finally, we add a few small-radix switches to allow connection of the row and column buses to the twiddle factor SRAM, the transpose unit, or one of the two data SRAMs.

To minimize complexity, we enforce some limitations on the connections. As mentioned previously, all of

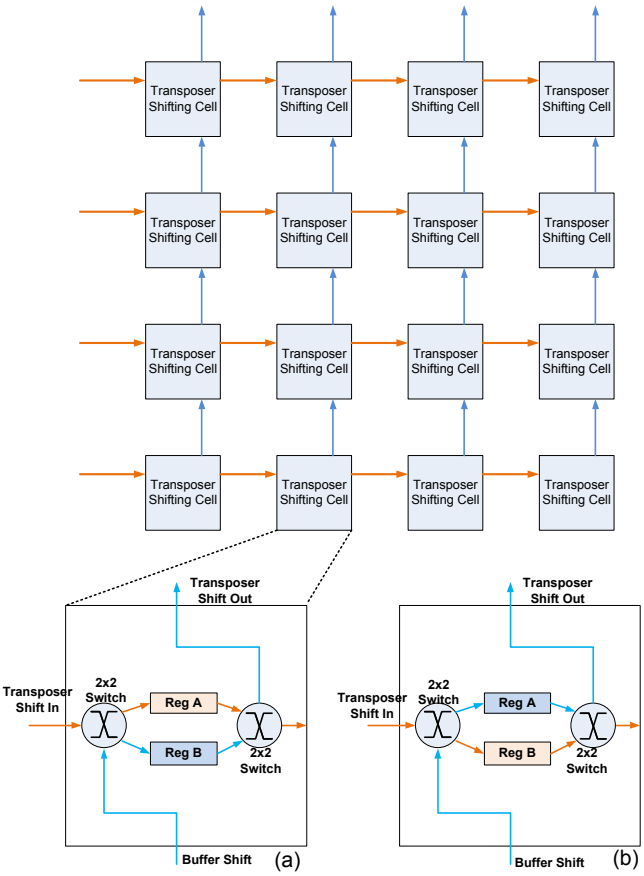

Fig. 12 Detail architecture of a 4×4 transposer, which is basically an array of shift registers combined in orthogonal direction for input and output and is double buffered.

these interfaces are bandwidth-matched, such that only 1:1 connectivity is possible. Only one of the two data SRAM banks is accessible to the transpose unit and local core during each computational step, with the other data SRAM bank used exclusively for transferring results from the previous transform out and pre-loading input data for the next transform. The SRAM holding the twiddle factors does not require connectivity to the transpose unit, since twiddle factors are only used in a point-wise distributed fashion. Finally, the core can connect to either the transpose unit or (one of the) local data SRAM banks, but not both at the same time (i.e., if both row and column buses are performing external data transfers at any one time, one set has to be reading from the twiddle factor SRAM.)

Rather than incurring the complexity of a general-purpose $P \times P$ crossbar, the transpose unit proposed here can be considered (at a high level) to be a set of cross-coupled row and column FIFOs that provide full-bandwidth bidirectional transposition between the P data SRAMs and the P cores. Fig. 12 illustrates the transposer configuration for one data transfer direction (reading data in on the rows and writing data out on

the columns). Data transfer in the other direction (not shown) is entirely analogous.

For a configuration with P cores and P SRAMs, the transpose unit will contain two sets of $P \times P$ registers. Each pair of registers is grouped in a "shifting cell". The shifting cells are arranged in a two-dimensional array with P bidirectional row ports and P bidirectional column ports. When operating in row-to-column mode, each of the P registers in row i can only be written by input port i, while each of the registers in column j can only be read by output port j. The converse is true for column-to-row mode.

The shifting cells of the transposer use one set of the registers to buffer the input shifting data (as a FIFO along each row) and the other set to shift the previously buffered data out (as a FIFO along each column). Each of the 2×2 switches contains two 2:1 multiplexers. One of the switches selects the correct register for buffering the input and the other selects the other one to shift the output. In Fig. 12(a) we show a case where Reg A is used for buffering the input and Reg B is used to shift the previously buffered data out. Alternatively Fig. 12(b) is the case when Reg B is buffering and Reg A is shifting data out. By double-buffering the data in this way, we completely separate row and column transfers in time, so no dynamic switching is required and no conflicts can occur.

Consider the 1-D transform of a vector of length $N = N_1 \times N_2$ using the four-step method and P cores. Assuming the data is distributed across the cores by row, stage 1 must start with reading the columns of the array, transposing them, and sending them to the cores. By careful selection of the order in which the data is read, bandwidth can be maximized and latency minimized. E.g., for P cores, P elements at a stride of N_1/P (one element bound for each core) are fetched from each SRAM to be sent to the transpose unit. This is repeated P times with each repetition incrementing the row number of the data selected. This results in a $P \times P$ set of values in the transpose unit, which are then sent to the cores in transposed order. Thus for an array factored into $N = N_2 \times N_1$ (rows by columns), the first N_2 elements sent to each core constitute one complete column. At this point, each core can begin transforming its first column while additional columns are read from the transposer and buffered into core memory.

After the columns have been transformed, the data is returned in the reverse direction. The only difference is that the indices produced by the algorithm above are bit-reversed before being used to access the core memory, thus putting the data back in natural order as it is returned to the offcore memory.

For small problem sizes, all of the input data can be buffered in core memory before the transposer needs to be turned around to enable storing the data back to SRAM. For larger problem sizes, the buffering requires multiple phases of loads and stores through the transposer, but in all of these cases there are enough idle data transfer cycles (waiting for the transforms to complete) to completely hide the transposer turnaround cycles. Therefore, with the exception of a single row or column transfer to fill the pipeline and another to drain it, all data transfers are overlapped with computation.

8 Evaluation Results

In this section, we present performance, area and power estimates for LAFC-based multi-core FFT accelerators, and we compare our single and multi-core solution with other available single- and double-precision implementations on various platforms. We note that this is a technology evaluation, not a complete design, so the power and area estimates are approximations. We model all the FPUs, the registers, the busses, the in-core and off-core SRAMs, and the long-distance on-chip interconnects. We do not model the many small state machines required for control and sequencing of operations, address generators, bit-reversal for the addresses, off-chip interfaces, error detection/correction circuits, self-test circuits, etc. Given the statically predictable cycle counts for the small number of operating modes and transform sizes supported by this engine, we believe that a simple mostly-distributed control structure is possible, though the details will depend on the specific usage scenario and are beyond the scope of this paper. Although these functions contribute significantly to the design effort for a system, they are not expected to be first-order contributors to the performance, area, or power estimates that we consider here.

8.1 Performance Estimation

As discussed in Section 7.1, the PEs execute radix-4 operations at a nominal 83.3% utilization in the absence of overheads. Two classes of overheads will be considered here: the time required to execute the global twiddle factor multiplications in the four-step method, and the data transfer time that is not overlapped with computation (incurred when starting up and shutting down the pipelined row and column transforms in the four-step method).

In most cases, the larger of the two overheads is the time required to multiply the data by the global twiddle

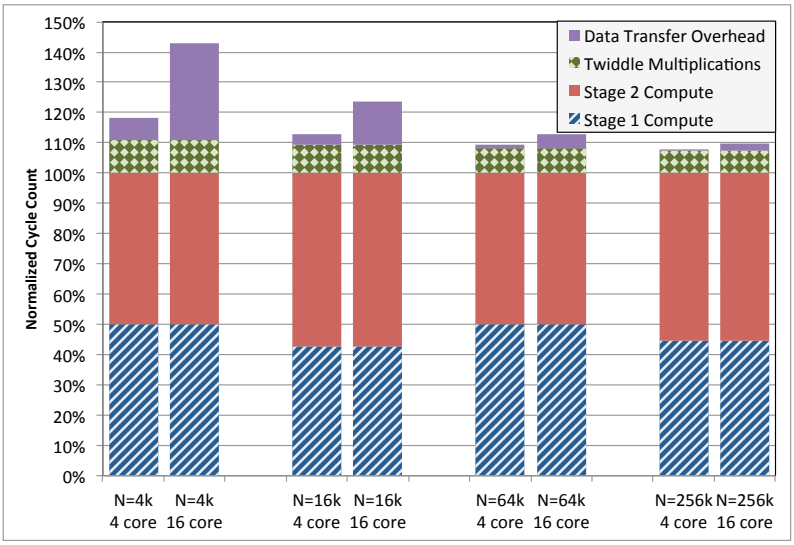

Fig. 13 Normalized cycle counts for four transform sizes on 4 and 16 cores. Cycle counts are normalized against the total compute cycles for the row and column transforms.

factors between the two transform steps in the four-step method. This is a linear overhead of $N/4$ cycles. As such, it does not change the asymptotic efficiency of the algorithm, but does change the effective utilization slightly. For problem sizes of $N = 4096$ or smaller, the core can perform the transform directly, so the four-step algorithm is not needed. For $N = 16384$, the extra cycles reduce the nominal utilization to 76.1%, with larger problems showing slightly less degradation due to the increase in the $N \log_2 N$ term.

In the single-core configuration, the overhead required to start up and drain the pipeline is very small. This is due to offcore SRAM latencies being low (1.5 - 2.5 ns depending on size), and because the parallelism allows almost all data transfers to be overlapped with computation. The worst case occurs for the smallest problem size requiring the four-step method, $N = 2^{14}$, which we decompose as 2^8 transforms of length 2^6 in the first step and 2^6 transforms of length 2^8 in the second step. At a bandwidth of 2 complex elements per cycle, loading the first column requires 32 cycles, after which data transfers are completely overlapped with computation. Once the final column transform has been computed, the copy-out of the final column can be overlapped with the loading of the first row. Loading the first row and the associated global twiddle factors (256 elements each) requires 256 cycles. Once the final row transform is completed, an additional 128 cycles are required to copy the final set of values to SRAM. The total overhead of $32 + 256 + 128 = 416$ cycles is less than 0.9% of the compute cycles, and is neglected here.

In addition to the overhead required for global twiddle factor multiplication, the multicore implementation contains additional latencies associated with transfers via the transpose unit. We assume a latency of 12 SRAM cycles (6 core cycles, or $L_D = 6$ ns) for the direct path between an SRAM and its local core. This remains mostly negligible compared to the $N_1/2$ core cycles required for any row or column transfer. For a transpose unit connecting P cores, we assume an additional $2P$ SRAM cycles (or P core cycles) of latency – P SRAM cycles to load a buffer in the transpose unit and P SRAM cycles to read that buffer in transposed order. This latency, L_P, will be incurred for both reading the initial column and writing the final column data. We do not attempt to overlap the writing of the final column data with the reading of the first row because that would require access to the off-core data SRAMs using both the direct path and the transpose path at the same time. By avoiding this case, the switching infrastructure can be simplified at a very small cost in performance (well under 1% for all cases except the most demanding case of N=4096 on 16 cores, where the performance penalty due to the added latency approaches 3%).

Given these assumptions, the core cycle count for the various cases can be condensed into the sum of the compute time T_c, the global twiddle factor multiplication time T_t, and the pipeline start/drain overhead time T_o. In these equations, $N = N_1 \times N_2$ is the problem size (with $N_1 \geq N_2$ to minimize the communication requirements in the second half of the 4-step method), P is the number of cores in use, L_D is the startup latency for a local SRAM access (6 ns), and L_P is the startup latency for an access through the transposer ($6 + P$ ns):

$$T_c = N/4 \times 24 \times \log_4 N/n_r^2$$
$$T_t = 4 \times N/n_r^2$$
$$T_o = 2 \times (2^{N_2}/2 + L_P) + 2 \times (2^{N_1}/2 + L_D)$$

PE Component	Estimate
SRAM	
SRAM Area	0.073 mm^2
SRAM Max Power	0.015 W
SRAM Actual Power	0.006 W
Floating-Point Unit	
FP Area	0.042 mm^2
FP Power	0.031 W
Register File	
RF Area	0.008 mm^2
RF Max Power	0.004 W
RF Actual Power	0.003 W
Broad-cast Buses	
Bus Area /PE	0.014 mm^2
Max Bus Power	0.001 W
PE Total	
PE Area	0.138 mm^2
PE Max Power	0.052 W
PE Actual Power (GEMM,FFT)	(0.037, 0.041) W
GFLOPS/W (GEMM, FFT)	(53.80, 40.50)
GFLOPS/Max W (GEMM, FFT)	(38.55, 32.12)
GFLOPS/mm^2 (GEMM, FFT)	(14.54, 12.11)
Power density	0.377 W/mm^2

Table 3 PE characteristics for a LAFC that can perform both linear algebra and FFT operations.

Fig. 13 shows the normalized cycle counts for problem sizes of N=4K, 16K, 64K, 256K (2^{12} to 2^{18}) for both 4-core and 16-core configurations. Each column is normalized against the sum of compute cycles for the row and column transforms. The relative compute cycle count for each transform step depends on the choice of factorization. The factorization is symmetric for $N = 2^{12}$ and $N = 2^{16}$, and biased towards longer transforms in stage 2 for $N = 2^{14}$ and $N = 2^{18}$. The global twiddle factor multiplication overhead is independent of the number of cores, decreasing from 11.1% for $N = 2^{12}$ to 7.4% for $N = 2^{18}$. The data transfer overhead includes the pipeline filling and draining at the beginning of each stage, which we do not overlap with computation. This overhead increases with the core count due to the extra latency required to buffer data through the (larger) transpose unit. For $N = 2^{12}$ on 4 cores, the data transfer overhead is a modest 6.9%, while on 16 cores the data transfer overhead is a marginally tolerable 31.9%. For larger problem sizes, the data transfer overhead drops rapidly (slightly faster than a linear decrease, as expected from the $N \log_2 N$ work requirement), with even 16 cores delivering better than 88% parallel efficiency for problems as small as $N = 2^{16}$.

8.2 Area and Power Estimation

The basic PE and core-level estimations of a LAFC in 45nm bulk CMOS technology were reported previously in [26], using CACTI to estimate the power and area consumption of memories, register files, look-up tables and buses. Power and area of the floating-point units use the measurements reported in [9].

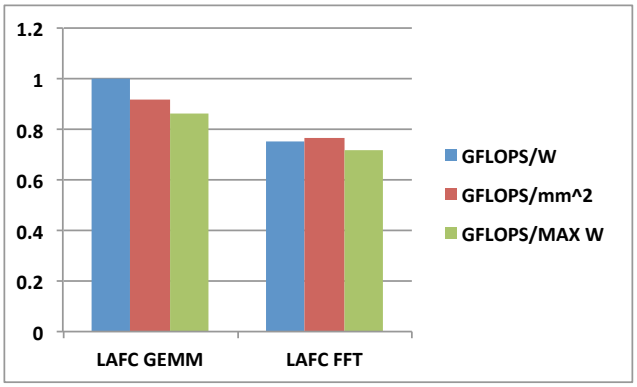

Fig. 15 LAFC Normalized Efficiency parameters for GEMM and FFT for the target applications normalized based on the LAC parameters at 45nm.

Here we review these area and power estimates for the core, then present our methodology and results for estimating area and performance for the multi-core extensions in more detail.

Core-level area and power estimates Table 3 summarizes the projected power and area consumption of PE components running at 1 GHz. For total efficiency, we report a pair of numbers of a LAFC either running GEMM or FFT. Fig. 14 plot the power and area breakdown of the LAFC design. The "actual" power considers the worst-case power when running either GEMM or FFT, while the "maximum" power breakdown shows the peak power that is used by the LAFC. We observe that the power consumption is dominated by the FP-MAC unit, with secondary contributions from the PE-local SRAMs. Furthermore, the area breakdown emphasizes that most of the PE area is occupied by the memory blocks.

Fig. 15 demonstrates the efficiency metrics of the LAFC when running GEMM or FFT. Note that in all cases, efficiency numbers are scaled by achievable utilization. Efficiency numbers are normalized against the original LAC design optimized for GEMM operations only. We can observe that the LAFC has lower efficiency when considering maximum power and area. However, since the leakage power consumption of the SRAM blocks is negligible, the LAFC maintains the power efficiency of a pure LAC when running GEMM. By contrast, FFT operation results in 20% efficiency drop due to extra communication overhead.

Multi-core power and area estimates For the multicore system, we must model the overhead of an additional level in memory hierarchy, plus the overhead of the transpose unit required for global communication. The

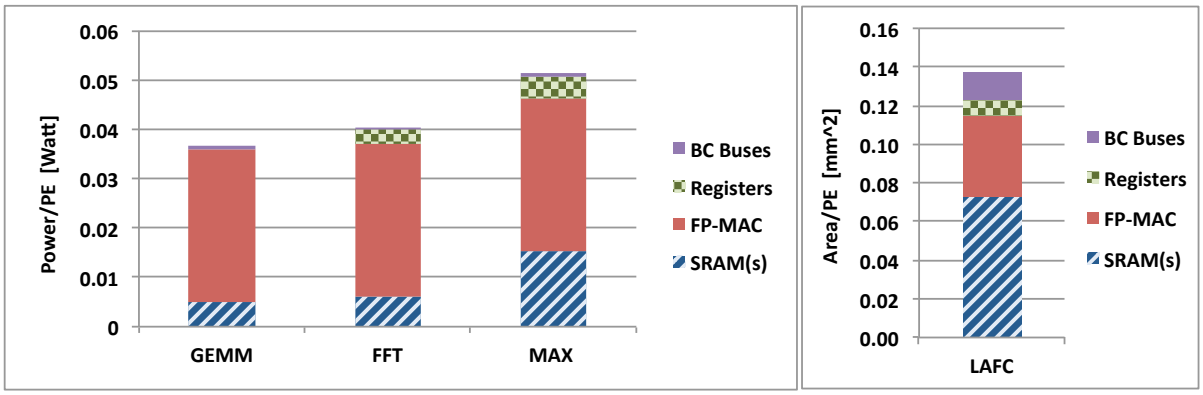

Fig. 14 LAFC PE parameters at 1GHz and 45 nm technology: Actual and Maximum power consumption (left). Area (right).

Problem Size	total cycles	SRAM Dyn (W)	SRAM leak(W)	SRAM total(W)	Cores (W)	Wires + Xposer (W)	Total (W)	GFLOPS/ W	GFLOPS/ (mm^2)	GFLOPS	Util
Double Precision @45nm											
4 Cores											
4K	2720	1.097	0.233	1.330	2.640	0.011	3.981	22.69	1.02	90.35	71%
16K	12128	0.984	0.233	1.217	2.640	0.010	3.867	24.45	1.06	94.56	74%
64K	53792	1.035	0.233	1.268	2.640	0.009	3.918	24.88	1.10	97.47	76%
256K	238880	0.933	0.233	1.166	2.640	0.008	3.814	25.90	1.11	98.76	77%
16 Cores											
4K	828	1.828	0.251	2.080	10.560	2.812	15.452	19.21	1.97	296.81	58%
16K	3324	1.821	0.251	2.073	10.560	2.802	15.435	22.35	2.29	345.03	67%
64K	13884	1.744	0.251	1.996	10.560	2.683	15.239	24.78	2.50	377.62	74%
256K	60732	1.861	0.251	2.112	10.560	2.454	15.126	25.68	2.57	388.48	76%
Single Precision @45nm											
4 Cores											
4K	2720	0.357	0.114	0.471	1.240	0.006	1.716	52.64	2.10	90.35	71%
16K	12128	0.320	0.114	0.434	1.240	0.005	1.679	56.32	2.20	94.56	74%
64K	53792	0.337	0.114	0.451	1.240	0.005	1.695	57.49	2.27	97.47	76%
256K	238880	0.304	0.114	0.417	1.240	0.004	1.661	59.45	2.30	98.76	77%
16 Cores											
4K	828	0.629	0.123	0.752	4.960	1.406	7.118	41.70	4.06	296.81	58%
16K	3324	0.626	0.123	0.749	4.960	1.401	7.110	48.53	4.72	345.03	67%
64K	13884	0.600	0.123	0.723	4.960	1.342	7.024	53.76	5.17	377.62	74%
256K	60732	0.640	0.123	0.763	4.960	1.227	6.950	55.90	5.31	388.48	76%

Table 4 Estimated power, performance, and area metrics for 4-core and 16-core configurations performing 1D FFTs on various vector lengths at 45nm.

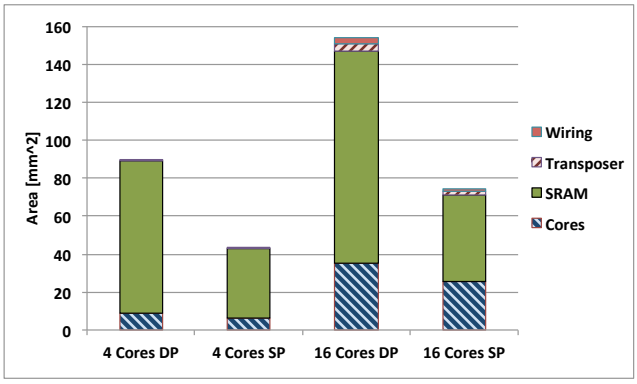

Fig. 16 Area breakdown for 4 and 16 cores DP and SP configurations at 45nm.

result of the power and area analysis are summarized for four different problem sizes in Table 4. We examine 4-core and 16-core solutions using the algorithmic choices previously presented in Table 1. We chose an aggregate off-core SRAM of 12 MiB for all the multicore configurations. This can hold the data, twiddle factors, and buffers for overlapping off-chip data transfer for problem sizes up to N=256K.

The off-core memory power is estimated using SRAM leakage power and energy per access from the CACTI simulations, combined with the access counts for performing a complete FFT of a given size, with the average power computed as the total energy used divided by the cycle count estimates from Section 8.1. For each configuration we adjusted the off-core SRAM configurations until CACTI reported minimum cycle times of less than 0.5 ns (supporting operation at twice the nominal 1 GHz frequency of the cores), and used the corresponding area, leakage power, and energy per access in the subsequent analysis.

For the four-core system, synthesis of the transpose unit as a set of FIFOs resulted in a projection of negligible area and power consumption ($< 0.1\%$ and $< 0.3\%$ of the total, respectively). Due to the simple geometry

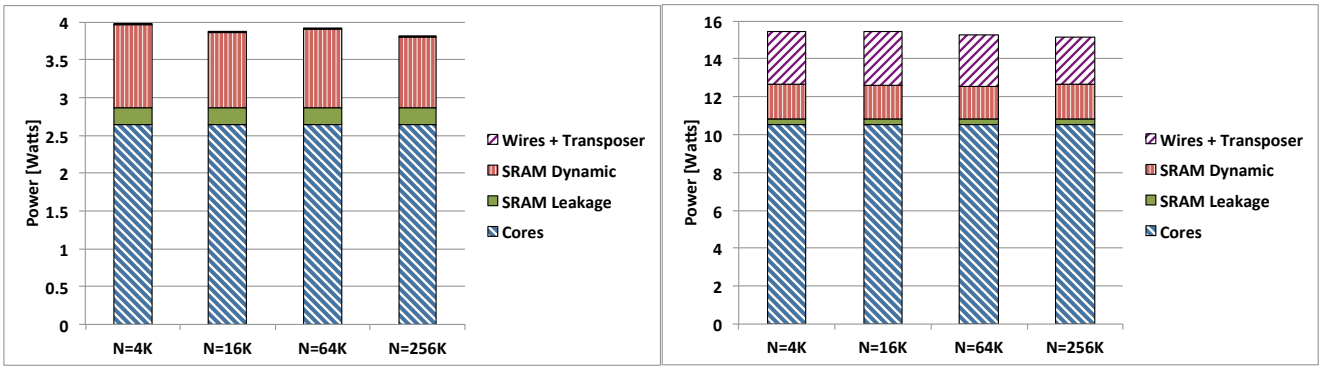

Fig. 17 Power breakdown of the double-sprecision multi-core solution with on-chip SRAMs to fit 256K size FFT with 12 MBytes of on-chip SRAM. 4-LAFC (left), 16-LAFC (right).

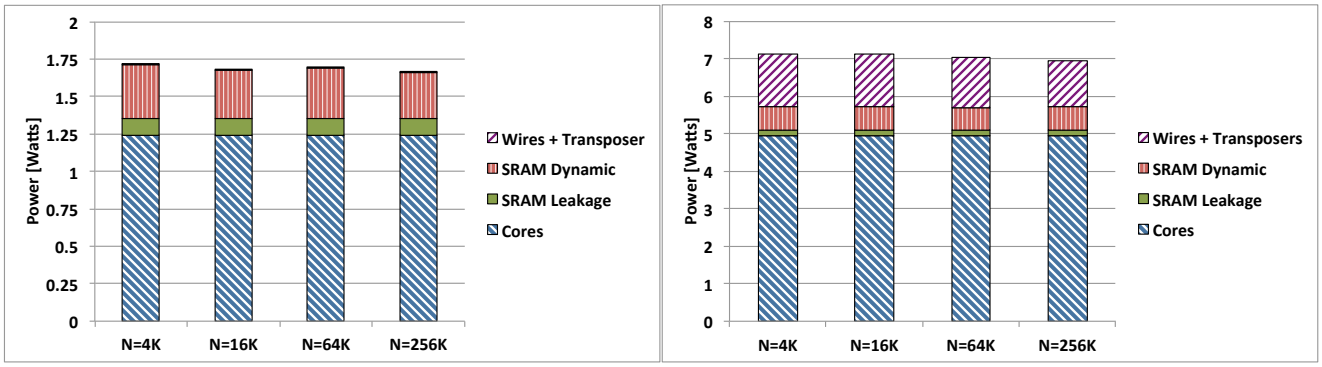

Fig. 18 Power breakdown of the single-precision multi-core solution with on-chip SRAMs to fit 256K size FFT with 6 MBytes of on-chip SRAM. 4-LAFC (left), 16-LAFC (right).

of a 2×2 array, we assume that the four "core plus private off-core SRAM" blocks can be organized so that the transposer is physically close to the data ports for the SRAMs and cores, so wire area and power are assumed to be small and are not separately computed.

For the 16-core system, synthesis of the transpose unit as a set of FIFOs also resulted in fairly low area and power consumption estimates, but it was not obvious that current technology would support the wiring density and bandwidths required. Recent work [28], however, has demonstrated a full crossbar operating at 4.5 Tb/s and 3.4 Tb/s/W fabricated in 45 nm technology – suggesting that the 4.096 Tb/s peak bandwidth required for the 16-core transposer in this project is feasible. Given that concrete design with accompanying hardware measurements, we chose to estimate our energy per transfer and area using the 3.4 Tb/s/W and 4.06 mm^2 values from [28].

We estimate the area and power of the long-distance wires using values from the same implementation, as described in [29]. Given the overall chip size of approximately 150 mm^2, we estimate that a 4×4 array of cores with a centrally located transpose unit will yield an average wire length of 4 mm. The long-distance wire area estimate assumes no overlap between the global wires and the cores or SRAMs, with 200 nm wire spacing and 128 wires between each core and the transposer. For power, this methodology predicts an average long-wire transfer energy of 2 pJ/bit. This value is almost an order of magnitude larger than the energy required inside the transpose unit itself, though the sum of the transposer and wire power remains under 20% of the total average power.

Fig. 16 demonstrates the area breakdown of the multi-core chip with 4 and 16 cores for single and double precision units. For the 4-core configuration the wiring and transposer area is negligible, leaving the SRAMs as the main consumer of the real estate on the chip – occupying 90% and 85% of the area in the double precision and the single precision configurations, respectively. When the number of cores is increased to 16 (with the off-core SRAM remaining at 12 MiB in total), the area occupied by the cores increases to 22% and 35% of the total area for double and single precision configurations respectively, while the transposer and long-distance wires add about 5% to the chip area.

Figs. 17 and 18 and show the power consumption breakdown for the four-step algorithm on a multi-core

architecture using double-precision and single-precision arithmetic respectively. The overall pattern for single-precision is very similar to the double-precision case at slightly less than half of the total power consumption. Figs. 17(left) and 18(left) report the 4-core configuration. We can see a slight decrease in power consumption as the problem size increases for both single and double -precision. However, for N=64K, the design consumes more power as a result of changing the algorithm and the extra load of global twiddle factor transfers between cores and SRAMs. Despite the large increase in data communication requirement, the cores remain the main sources of power consumption in the 16-core designs, as shown in Figs. 17 (right) and 18(right). This emphasizes that the memory hierarchy around the cores is carefully designed to consume low energy.

As shown in Table 4 and Fig. 13, the 4-core solution delivers excellent scaling over the considered problem sizes with nominal performance of 90.4 to 98.8 GFLOPS and 22.7 to 25.9 GFLOPS/W double-precision efficiency. The 16-core configuration suffers some scalability losses for the smallest two problem sizes, with speedups of 3.3× and 3.6× compared to the 4-core configuration, and with correspondingly reduced energy efficiency. For the larger two problem sizes the 16-core configuration delivers better than 96.8% parallel efficiency, with the double-precision case showing energy efficiency almost identical to that of the 4-core configuration. This improvement in scaling with problem size is expected, but the negligible power/performance penalty for the 16-core double precision case is somewhat surprising. The explanation lies in the difference in SRAM configurations. The 16-core case uses 48 off-core SRAMs of 0.25 MiB each, while the 4-core case uses 12 off-core SRAMs of 1.0 MiB each. The smaller SRAMs are more efficient, requiring almost exactly half the energy per access of the larger SRAMs. This energy savings almost exactly cancels the extra energy expenditure required for the transposer and long wires in the 16-core double precision case. For the single-precision case the reduction in SRAM power is smaller than the long distance wire power overhead leading to an overall reduction in power efficiency of up to about 7% for the larger two problem sizes.

In summary, the designs with 4 and 16 cores deliver in excess of 90 and 300 GFLOPS double-precision performance while consuming less than 4 and 13 Watts of power, respectively. This includes power for the cores, the off-core SRAMs in active use, the off-core SRAMs used to pre-load and post-store the next data set, and the transposer and long wires.

8.3 Comparison to Other Processors

Table 5 provides comparisons of estimated performance, area, and power consumption between our proposed design and several alternative processors – including a state-of-the-art general-purpose processor [33], a low-power ARM processor [4], a version of the IBM Cell processor [16], and a recent NVIDIA GPGPU [32]. In this table, we limit the comparison to double-precision 1D FFT performance for problem sizes that fit into on-chip SRAM or cache. All area and power (but not performance) estimates are scaled to 45nm technology and (to the extent possible) include only the area and power associated with the cores and SRAMs or caches that are in actual use for each problem size considered. Although these comparisons are necessarily coarse, we find that in each case, the proposed FFT engine provides at least an order of magnitude advantage in efficiency and significant advantages in performance per unit area (which is much harder to interpret cleanly due to the dominance of SRAMs in the area requirements).

We were only able to find limited published data for specialized FFT cores configured for double-precision floating-point arithmetic, so we prepared power and area estimates for our system configured for single -precision floating-point operation. We compare these results to published single- precision alternatives in Table 6. It is important to note that three of the four designs used for comparison are streaming implementations optimized for single-length transforms of fixed size elements. For these cases, the most appropriate comparison is the single-core LAFC without additional off-chip SRAM (the first entry in Table 6). A single hybrid, in-core LAFC is estimated to deliver a respectable 62% of the GFLOPS/W of the specialized ASIC system (configured for streaming transforms of a fixed length of 4096 elements), though with much poorer area efficiency – most likely due to the relatively large on-core SRAMs retained in our LAFC core. The FPGA reference in the table uses the same design methodology as the ASIC, but is optimized for streaming transforms of length 1024. Despite the simplifications allowed by this specialization, our proposed design is projected to achieve more than an order of magnitude better performance per unit energy and performance per unit area than this highly optimized FPGA design.

Two additional comparisons are included in the single precision table. The Godson-3B result is also specialized for streaming FFTs of length 1024, but has the highest performance per Watt of any general-purpose processor that we were able to find in our literature survey. The Cell processor result is somewhat dated, but is included because it contains both parallelization

Platform Running FFT	Problem fits in	$/SRAM [KBytes]	Peak GFLOPS	FFT Eff GFLOPS	Power [Watt]	Area [mm^2]	GFLOPS/ Watt	GFLOPS/ mm^2	Util
LAFC core	On-core SRAM	320	32.0	26.7	0.66	2.2	40.5	12.12	83.3%
LAFC core	On-chip SRAM	2048	32.0	26.7	0.96	13.2	27.7	2.01	83.3%
Xeon E3-1270 core	L1 cache	32	27.2	18.0	28	27.7	0.64	0.65	66.2%
Xeon E3-1270 core	L2 cache	256	27.2	12.0	28	36.6	0.43	0.33	44.1%
ARM A9 (1 GHz)	L1 cache	32	1.0	0.6	0.28	6.3	2.13	0.09	60.0%
LAFC 4-core	On-chip SRAM	12288	128	98.8	3.8	88.9	25.95	1.11	77.2%
LAFC 16-core	On-chip SRAM	12288	512	388.5	12.67	146.9	30.66	2.65	75.9%
PowerXCell 8i SPE	SPE local RAM	2048	102.4	12.0	64	102	0.19	0.12	11.7%
NVIDIA C2050	L1+L2 cache	1728	515.2	110.0	150.00	529.0	0.73	0.21	21.3%
Xeon E3-1270 4-core	L3 cache	8192	108.8	39.1	91.73	231.4	0.43	0.17	35.9%

Table 5 Comparison between the proposed single- and multi-core designs and several alternatives for cache-contained double-precision FFTs scaled to 45nm.

Platform Running FFT	Problem fits in	$/SRAM [KBytes]	Peak GFLOPS	FFT Eff GFLOPS	Power [Watt]	Area [mm^2]	GFLOPS/ Watt	GFLOPS/ mm^2	Util
LAFC 1-core	On-core SRAM	320	32.0	26.7	0.31	1.6	86.0	16.67	83.3%
LAFC 1-core	Off-core SRAM	1024	32.0	26.7	0.43	7.7	62.0	3.47	83.3%
LAFC 4-core	On-chip SRAM	6144	128	98.76	1.66	43.0	59.4	2.30	77.2%
LAFC 16-core	On-chp SRAM	6144	512	388.48	6.95	73.0	55.9	5.32	75.9%
ASIC [19]	(streaming,N=4096)	-	(N/A)	1200	8.7	10	137.9	125	N/A
FPGA [19, 6]	(streaming,N=1024)	-	(N/A)	380	60	380	6.3	1.0	N/A
Godson-3B [18]	(streaming,N=1024)	4096	256	139	28	144	5.0	0.97	54.3%
Cell BE [10]	SPE local RAM	2048	192	48.2	20	55	2.4	0.87	25.1%

Table 6 Comparison between the proposed Hybrid core modified for single-Precision (32-bit IEEE floating-point) operation, two highly optimized special-purpose FFT designs proposed in [19], the Godson-3B processor [18], and a highly optimized implementation for the Cell BE [10].

across the eight Synergistic Processing Elements of the Cell Processor and a problem size (N=64K) that is considerably larger than those typically found in the literature. All of the configurations of the single-precision LAFC processor are projected to deliver more than an order of magnitude better performance per Watt than these two options, as well as maintaining a significant advantage in performance per unit area (despite having significantly more on-chip SRAM).

9 Summary, Conclusions and Future Work

Beginning with a high-efficiency double-precision floating-point core capable of being configured for either linear algebra or FFT operations, we demonstrate how a combination of careful algorithm analysis with judiciously chosen data-path modifications allowed us to produce a multicore FFT architecture with excellent parallel scaling and minimal degradation in power efficiency and area efficiency of the single-core design.

We project nominal FFT performance of up to 388 GFLOPS for the 16-core configuration running at 1.0 GHz on problem sizes that can fit in on-chip SRAM. For a 45 nm technology target, the power efficiency varies between 25 GFLOPS/W and over 30 GFLOPS/W – more than an order of magnitude better than any available double-precision alternative. For single-precision operation, we project power efficiencies of at least 55 GFLOPS/W, which is within a factor of three compared to state of the art special-purpose FFT engine customized for single-sized (N=4096) transforms.

Area efficiency is dominated primarily by the off-core memory required for the four-step algorithm used to compute large transforms and to compute transforms using multiple cores. Even with this overhead, the area efficiency of 1.0 GFLOPS/mm^2 to 2.6 GFLOPS/mm^2 is significantly better than general-purpose solutions or FPGA implementations.

Significantly, our technology evaluation shows that the efficiency of the four-core design is comparable to the efficiency of the single-core, while the increased efficiency of smaller off-core SRAMs (per core) allows the 16-core solution to deliver better energy efficiency than either the single-core or four-core versions for all except the smallest problem sizes considered.

Ongoing work includes hardware/software co-design to permit use of memory technologies that require exploitation of locality for effective use (such as eDRAM), the additional complexities involved in using off-chip DRAM (stacked or traditional), and analysis of the performance, energy, and area tradeoffs involved in computing some or all of the (double precision) global twiddle factors rather than storing all of them.

We also plan to look into additional specialization of the FFT designs on the same core to achieve better utilization and efficiency using customized floating-point units.

Acknowledgements Authors wish to thank John Brunhaver for providing synthesis results for the raw components of the Transposer.

References

1. Akin, B., Milder, P.A., Franchetti, F., Hoe, J.C.: Memory bandwidth efficient two-dimensional fast Fourier transform algorithm and implementation for large problem sizes. In: Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, FCCM '12, pp. 188–191. IEEE (2012)
2. Bailey, D.H.: FFTs in external or hierarchical memory. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pp. 234–242. ACM (1989)
3. Bergland, G.: Fast Fourier transform hardware implementations–an overview. IEEE Transactions on Audio and Electroacoustics **17**(2), 104 – 108 (1969)
4. Blake, A., Witten, I., Cree, M.: The fastest Fourier transform in the south. Signal Processing, IEEE Transactions on **61**(19), 4707–4716 (2013)
5. Cheney, M., Borden, B., of the Mathematical Sciences, C.B., (U.S.), N.S.F.: Fundamentals of radar imaging. CBMS-NSF regional Conference series in applied mathematics. SIAM, Philadelphia, PA, USA (2009)
6. Chung, E.S., Milder, P.A., Hoe, J.C., Mai, K.: Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In: 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-43, pp. 225–236. IEEE Computer Society, Washington, DC, USA (2010)
7. Cooley, J., Tukey, J.: An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation **19**(90), 297–301 (1965)
8. Frigo, M., Johnson, S.: The design and implementation of FFTW3. Proceedings of the IEEE **93**(2), 216–231 (2005)
9. Galal, S., Horowitz, M.: Energy-efficient floating point unit design. IEEE Transactions on Computers **PP**(99) (2010)
10. Greene, J., Pepe, M., Cooper, R.: A parallel 64k complex FFT algorithm for the IBM/Sony/Toshiba Cell broadband engine processor. In: Conference on the Global Signal processing Expo (2005)
11. Hemmert, K.S., Underwood, K.D.: An analysis of the double-precision floating-point FFT on FPGAs. In: Proceedings of the 2005 IEEE 13th International Symposium on Field-Programmable Custom Computing Machines, FCCM '05, pp. 171–180 (2005)
12. Ho, C.H.: Customizable and reconfigurable platform for optimising floating-point computations. Ph.D. thesis, University of London, Imperial College of Science, Technology and Medicine, Department of Computing (2010)
13. Jain, S., Erraguntla, V., Vangal, S., Hoskote, Y., Borkar, N., Mandepudi, T., Karthik, V.: A 90mW/GFlop 3.4GHz reconfigurable fused/continuous multiply-accumulator for floating-point and integer operands in 65nm. In: 23rd International Conference on VLSI Design, 2010. VLSID '10., pp. 252–257 (2010)
14. Kak, A., Slaney, M.: Principles of computerized tomographic imaging. Classics In Applied Mathematics. SIAM, Philadelphia, PA, USA (2001)
15. Karner, H., Auer, M., Ueberhuber, C.W.: Top speed FFTs for FMA architectures. Tech. Rep. AURORA TR1998-16, Institute for Applied and Numerical Mathematics, Vienna University of Technology (1998)
16. Kistler, M., Gunnels, J., Brokenshire, D., Benton, B.: Petascale computing with accelerators. In: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '09, pp. 241–250. ACM, New York, NY, USA (2009)
17. Kuehl, C., Liebstueckel, U., Tejerina, I., Uemminghaus, M., Witte, F., Kolb, M., Suess, M., Weigand, R., Kopp, N.: Fast Fourier Transform Co-processor (FFTC), towards embedded GFLOPs. In: Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, vol. 8539 (2012)
18. Li, L., Chen, Y.J., Liu, D.F., Qian, C., Hu, W.W.: An FFT performance model for optimizing general-purpose processor architecture. Journal of Computer Science and Technology **26**(5), 875–889 (2011)
19. Milder, P., Franchetti, F., Hoe, J.C., Püschel, M.: Computer generation of hardware for linear digital signal processing transforms. ACM Transactions on Design Automation of Electronic Systems **17**(2) (2012)
20. Mou, S., Yang, X.: Design of a high-speed FPGA-based 32-bit floating-point FFT processor. In: Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on, vol. 1, pp. 84–87 (2007)
21. Pedram, A., van de Geijn, R., Gerstlauer, A.: Codesign tradeoffs for high-performance, low-power linear algebra architectures. IEEE Transactions on Computers, Special Issue on Power efficient computing **61**(12), 1724–1736 (2012)
22. Pedram, A., Gerstlauer, A., van de Geijn, R.: On the efficiency of register file versus broadcast interconnect for collective communications in data-parallel hardware accelerators. In: Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 19–26 (2012)
23. Pedram, A., Gerstlauer, A., Geijn, R.A.: A high-performance, low-power linear algebra core. In: Proceedings of the 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP '11, pp. 35–42. IEEE Computer Society, Washington, DC, USA (2011)
24. Pedram, A., Gerstlauer, A., van de Geijn, R.A.: Floating point architecture extensions for optimized matrix factorization. In: Proceedings of the 2013 IEEE 21st Symposium on Computer Arithmetic, ARITH '13. IEEE (2013)
25. Pedram, A., Gilani, S.Z., Kim, N.S., Geijn, R.v.d., Schulte, M., Gerstlauer, A.: A linear algebra core design for efficient level-3 BLAS. In: Proceedings of the 2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors, ASAP '12, pp. 149–152. IEEE Computer Society, Washington, DC, USA (2012)
26. Pedram, A., McCalpin, J., Gerstlauer, A.: Transforming a linear algebra core to an FFT accelerator. In: Proceedings of the 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 175–184 (2013)
27. Pereira, K., Athanas, P., Lin, H., Feng, W.: Spectral method characterization on FPGA and GPU accelerators. In: Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on, pp. 487–492 (2011)
28. Satpathy, S., Sewell, K., Manville, T., Chen, Y.P., Dreslinski, R., Sylvester, D., Mudge, T., Blaauw, D.:

A 4.5Tb/s 3.4Tb/s/W 64x64 switch fabric with self-updating least-recently-granted priority and quality-of-service arbitration in 45nm CMOS. In: Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International, pp. 478–480 (2012)
29. Satpathy, S., Sylvester, D., Blaauw, D.: A standard cell compatible bidirectional repeater with thyristor assist. In: VLSI Circuits (VLSIC), 2012 Symposium on, pp. 174–175 (2012)
30. Swartzlander Jr., E.E., Saleh, H.H.: FFT implementation with fused floating-point operations. IEEE Transactions on Computers **61**(2), 284–288 (2012)
31. Varma, B.S.C., Paul, K., Balakrishnan, M.: Accelerating 3D-FFT using hard embedded blocks in FPGAs. VLSI Design, International Conference on pp. 92–97 (2013)
32. Wu, D., Zou, X., Dai, K., Rao, J., Chen, P., Zheng, Z.: Implementation and evaluation of parallel FFT on engineering and scientific computation accelerator (ESCA) architecture. Journal of Zhejiang University-Science C **12**(12) (2011)
33. Yuffe, M., Knoll, E., Mehalel, M., Shor, J., Kurts, T.: A fully integrated multi-CPU, GPU and memory controller 32nm processor. In: Proceedings of the 2011 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). IEEE (2011)
34. Van Zee, F.G., van de Geijn. FLAME Working Note #66, R.A.: BLIS: A framework for generating BLAS-like libraries. Technical Report TR-12-30, The University of Texas at Austin, Department of Computer Sciences (2012)
35. Zhang, Z., Wang, D., Pan, Y., Wang, D., Zhou, X., Sobelman, G.: FFT implementation with multi-operand floating point units. In: ASIC (ASICON), 2011 IEEE 9th International Conference on, pp. 216–219 (2011)