Transforming A Linear Algebra Core to An FFT Accelerator

Ardavan Pedram*, John McCalpin†, Andreas Gerstlauer*
*Department of Electrical and Computer Engineering,†Texas Advanced Computing Center
The University of Texas at Austin
*{ardavan,gerstl}@utexas.edu, † mccalpin@tacc.utexas.edu

***Abstract*—This paper considers the modifications required to transform a highly-efficient, specialized linear algebra core into an efficient engine for computing Fast Fourier Transforms (FFTs). We review the minimal changes required to support Radix-4 FFT computations and propose extensions to the micro-architecture of the baseline linear algebra core. Along the way, we study the critical differences between the two classes of algorithms. Special attention is paid to the configuration of the on-chip memory system to support high utilization. We examine design trade-offs between efficiency, specialization and flexibility, and their effects both on the core and memory hierarchy for a unified design as compared to dedicated accelerators for each application. The final design is a flexible architecture that can perform both classes of applications. Results show that the proposed hybrid FFT/Linear Algebra core can achieve 26.6 GFLOPS/S with a power and efficiency of 40 GFLOPS/W and 12 GFLOPS/mm^2, respectively.**

I. Introduction

Basic Linear Algebra Subroutines (BLAS) and Fast Fourier Transforms (FFTs) are two of the most important classes of algorithms in the computational sciences. General Matrix Multiplication (GEMM) is the primary component of the level-3 BLAS and of most dense linear algebra algorithms (and many sparse/structured linear algebra algorithms), which in turn have applications in virtually every area of computational science. With its high ratio of computation to data motion and its balanced use of addition and multiplication, GEMM typically provides the opportunity to demonstrate the maximum sustainable floating-point computation rate of a computer system.

By contrast, FFTs are fundamentally linked to the underlying mathematics of many areas of computational science. They 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 [1] and computerized tomographic imaging [2]. 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. Unlike GEMM, 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.

While GEMM is a straightforward kernel with simple, predictable data access patterns, the FFT provides more 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 includes 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, so the "balanced" FPUs on most current architectures will be under-utilized.

For both the GEMM and FFT algorithms, application-specific designs have been proposed that promise orders of magnitude improvements in power/area efficiency relative to general purpose processors [3], [4]. However, each of these have been isolated and dedicated design instances limited to one algorithm. With full-custom design increasingly becoming cost-prohibitive, there is a need for solutions that have enough flexibility to run a range of operations at the efficiency of full-custom designs. In this paper, we analyze the similarities between algorithms and show how one might transform an optimized GEMM core to an FFT core. We consider whether a combined core that can perform either operation efficiently is practical, and analyze the loss in efficiency required to achieve this flexibility.

Our starting point is a Linear Algebra Core (LAC) that we developed in previous work [5]. The core design and its efficiency were originally derived for GEMM operations. Applying minimal extensions, we showed how a LAC can support the full range of level-3 BLAS [6] and matrix-factorizations [7] with minimal loss in efficiency. In this paper, we investigate further extensions of the LAC to also support FFTs.

We begin by exploring FFT algorithms that may be suitable for the baseline LAC architecture. After evaluating LAC limitations and trade-offs for possible solutions, we introduce an "FFT core" that we have optimized for FFTs over a wide range of vector lengths. While optimized for performing FFTs, this core is based on a minimal set of modifications to the existing LAC architecture. We then take similarities between the original LAC and the FFT-optimized design to introduce a flexible, hybrid design that can perform both of these applications efficiently. Comparing both full-custom designs with our proposed hybrid core, we demonstrate the costs of flexibility versus efficiency.

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 for the

external memory hierarchy. As part of this process, we study multiple choices in 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 II we provide a brief overview of related work. Section III describes the conventional and Fused Multiply Add (FMA) Radix-2 and Radix-4 butterfly FFT algorithms. In Section IV, we describe the baseline LAC architecture. Section V describes the mapping of the FFT and Section VI studies the trade-off between different core configurations. In Section VII, we demonstrate the estimated design features and compare with some competing options. Finally, we conclude the paper in Section VIII.

II. 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 [8] – only four years after the publication of the foundational Cooley-Tukey algorithm [9].

The literature related to floating-point FFT hardware is considerably more sparse, especially for double-precision implementations. Important recent work includes the automatic generation of hardware FFT designs from high-level specifications [4]. These hardware designs can be used in either ASIC or FPGA implementations [10], but the published double-precision results for these designs are currently limited to FPGAs [11]. Hemmert and Underwood [12] provide performance comparisons between CPU and FPGA implementations of double-precision FFTs, and include projections of anticipated performance. Finally, 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 [10].

Performance of FFT algorithms varies dramatically across hardware platforms and software implementations, depending largely on the effort expended on optimizing data motion. General-purpose, microprocessor-based systems typically deliver poor performance, even with highly optimized implementations, because the power-of-2 strides of the FFT algorithms interact badly with set-associative caches, with set-associative address translation mechanisms, and with power-of-2-banked memory subsystems.

In Section VIII, we compare the performance, area, and power of our proposed designs with a sampling of floating-point FFT performance results on general-purpose processors, specialized computational accelerators, and GPUs.

III. FFT 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 Figure 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 into 83% utilization.

A modified, FMA-optimized butterfly is introduced in [13], 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 ($\omega = \omega_L^j$):

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

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 Figure 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 V. The number of loads also drops because only two of the three twiddle factors (ω_L^j and $\omega_L^2 j$) 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 implementa-

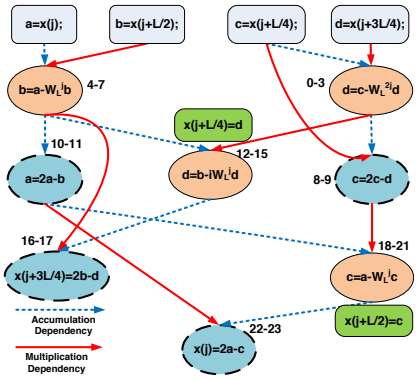

Fig. 1. DAG of the optimized Radix4 Butterfly using a fused multiply-add unit. Rectangles on top indicate the input data, solid nodes show complex computations with four FMA operations each, nodes with dashed lines show complex computations with two FMA operations each. The nodes are executed in an order that avoids data dependency hazards due to pipeline latencies, as shown by the start-finish cycle numbers next to each node.

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

IV. Baseline Linear Algebra Architecture

The microarchitecture of the baseline linear algebra core (LAC) is illustrated in Figure 2. The architecture and implementation optimize the rank-1 update operation that is the innermost kernel of parallel matrix multiplication [14]. This allows the implementation to achieve orders of magnitude better efficiency in power and area consumption than conventional general purpose architectures [3].

A. General Architecture

The LAC architecture consists of a 2D array of $n_r \times n_r$ Processing Elements (PEs), with $n_r = 4$ in Figure 2. Each PE has a double-precision Floating-Point Multiply-ACcumulate (FPMAC) unit with a local accumulator, and local memory (SRAM) storage divided into a larger single-ported and a smaller dual-ported memory. PEs on the same row/column are connected by low-overhead horizontal/vertical broadcast buses. LAC 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.

The FPMAC units perform the inner dot-product computations central to almost all level-3 BLAS operations. To achieve high performance and register-level locality, the LAC utilizes

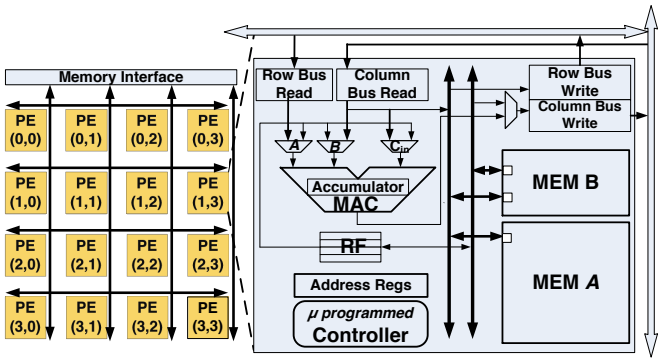

Fig. 2. Linear Algebra Core optimized for rank-1 updates. PEs that own the current column of $4 \times k_c$ matrix A and the current row of $k_c \times 4$ matrix B, write elements of A and B to the buses and the other PEs read them [3].

pipelined FPMAC units that can achieve a throughput of one *dependent* FPMAC operation per cycle [15]. This means that there is no data dependency hazard for floating point accumulations. Note that this is not the case in current general-purpose architectures [16], which require the use of multiple accumulators to avoid pipeline stalls.

V. FFT Algorithm Mapping

In this section, we show the details of mapping an FFT on the LAC along with the required modifications that need to be made to the existing core architecture. We start by focusing on small problems that fit in the local core memory. Then, we present solutions for bigger problems that do not fit in the local store.

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 LAC 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-4 FFT Algorithms on the PEs

In Section III we gave a description of regular and FMA optimized versions of the Radix-2 and Radix-4 butterfly operations. Here, we show the details of mapping such operations on the PEs. A Radix-2 operation takes six FMA operations. Performing Radix-2 operations in each PE, the LAC can perform 32-point FFTs, but can only hide the latency of FMA pipeline for FFT transforms with 64 or more points. The Radix-4 butterfly on the PE is more complicated due to data dependencies within the butterfly operation. Figure 1 shows the DAG of the Radix-4 butterfly. 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 FPMAC 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.

B. FFT on the Core

Here, we describe both Radix-2 and Radix-4 based FFTs on the LAC. We compare the computation and communication of these two options, including the bus access behavior.

1) Radix-2 based FFT: When PEs perform Radix-2 butterfly operations, each PE has to exchange one of its outputs with its neighbor of distance 2^0 (one) after the first stage. All PEs on the same row perform communication between PE_{2n} and PE_{2n+1}. After the second stage, PEs exchange outputs with those of neighbors at a distances of 2^1 (two). These PEs also fall on the same row of the 4×4 arrangement of the LAC. After the third stage, each PE exchanges its output with a PE that has a distance of 2^2 (four). In our architecture, with $n_r = 4$, this translates to adjacent neighbors on the same column. Finally, after the fourth stage, each PE switches its outputs with the PE that has a distance of 2^3 (eight). This also requires a column bus communication. In subsequent stages, the distances are multiples of $4^2 = 16$. In a 4×4 arrangement, these are mapped to the same PE. Therefore, there is no communication between PEs for these stages.

The shortcoming of performing Radix-2 butterflies on the PEs comes from a computation/communication imbalance. In stages two through four, broadcast buses are being used for exchanging data. For each exchange, 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%.

2) Radix-4 based FFT: The 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. Figure 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 $PEs_{(1,2,3)\times4^0}$, or PE_1=PE(0,1), PE_2=PE(0,2), and PE_3=PE(0,3) (See figure 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 or $PEs_{(4,8,12)} = PE_{(1,2,3)\times4^1}$, or PE_4=PE(1,0), PE_8=PE(2,0), and PE_{12}=PE(3,0). 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.

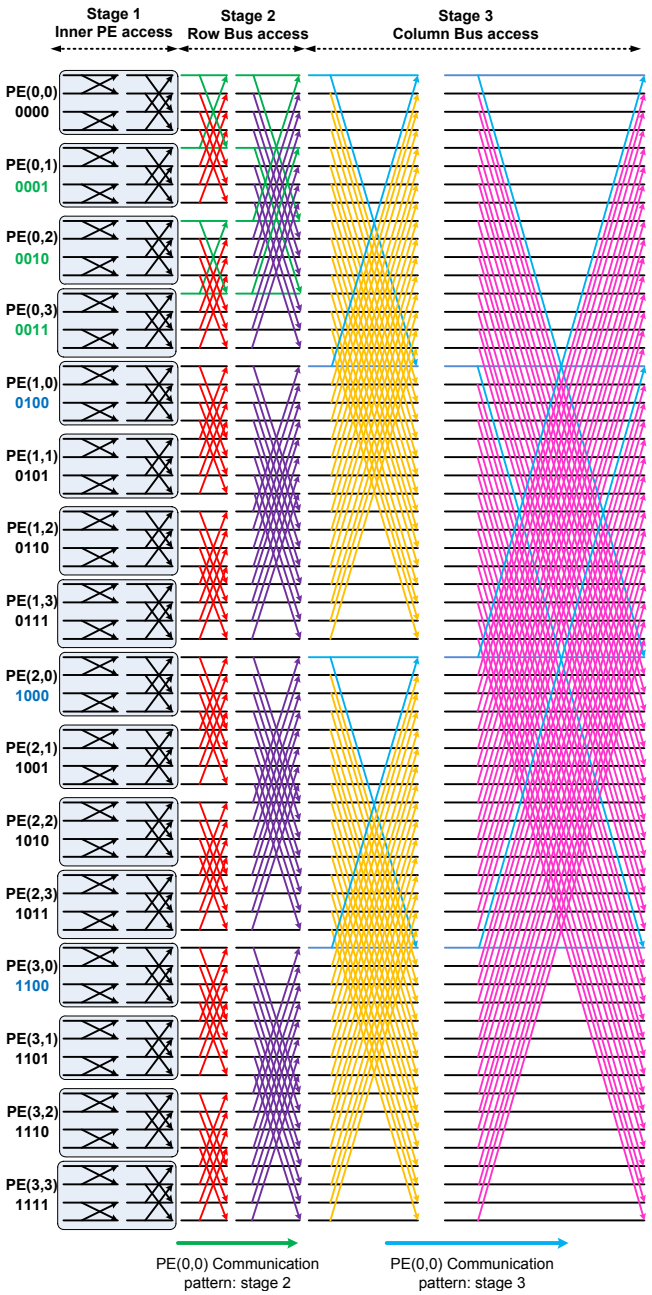

Fig. 3. 64 point FFT performed by 16 PEs in the core. Each PE is performing Radix-4 Butterfly operations. The access patterns for PE(0,0) are highlighted. Stage 2 only utilizes row-buses to perform data communications. Stage 3 only utilizes column-buses to perform data communications.

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

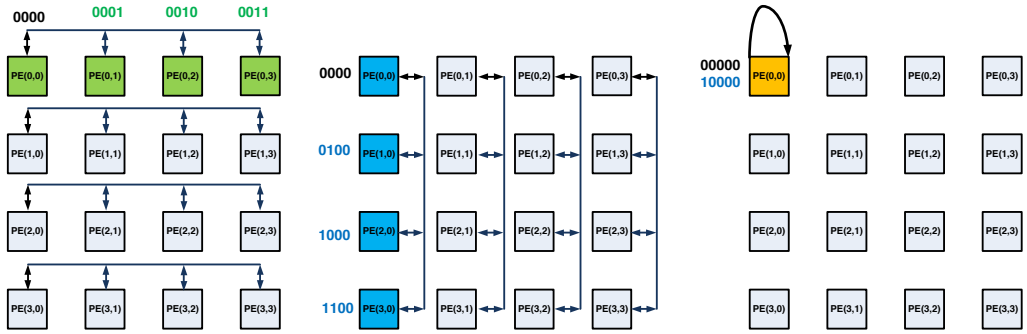

Fig. 4. Data communication access pattern between PEs of the LAC for Radix-4 FFT.

the column buses. The rest of data reordering is done by address replacement locally in each PE. Therefore as discussed in the next subsection, 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.

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.

C. FFT Memory Hierarchy for Bigger Transform Sizes

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. We will use 4096 as a "typical" value for the maximum size that can be transformed in PE-local memory, but we note that this is a configurable parameter.

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. This analysis is limited to on-chip (but off-core) memory. Considerations for off-chip memory are out of scope of this paper and are deferred to future work.

First, we note that the butterfly computations shown in Figure 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, multi-port SRAM arrays, capable of sourcing or sinking contiguous, strided, or random addresses at a rate matching or exceeding the bandwidth requirement of the core. Each of the solutions discussed below will be capable of handling the bit-reversal transformation, as well as any other data access patterns required.

1) Algorithm 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 [17]. For an FFT of length N, we split the length into the product of two integer factors, $N = N_1 N_2$. The 1D discrete Fourier transform can then be computed by the sequence: (1) Perform N_1 DFTs of size N_2; (2) Multiply the result by an array of complex roots of unity (called "twiddle factors"); (3) Perform N_2 DFTs of size N_1. For a core capable of performing transforms of up to N=4096, 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 of sizes.)

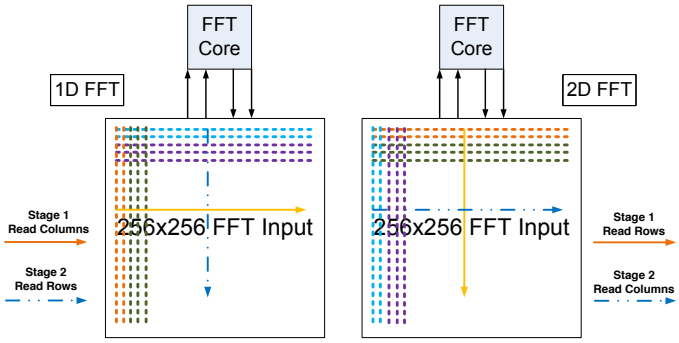

Fig. 5. Overview of data motion to/from the core for performing a 64K 1D FFT (left), and for a 256×256 2D FFT (right).

The overall data motion for the 1D FFT is shown in in the left panel of Figure 5. Assuming that the data begins in natural order, the first set of DFTs 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. In our design, 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. The results (which are in bit-reversed order) are written back to the SRAM array while applying bit-reversal permutation to restore them to natural order.

The second set of DFTs in the four-step algorithm occur on naturally contiguous data, but the "in-place" transforms of the first step operated on transposed data and left the data in transposed order, so a second transposition is required. This second transposition can be included in the writeback of the results from the first set of DFTs, or it can be included while reading the data for the second set of DFTs. The SRAM

FFT $N \times N$	2D No-Ov	2D Ov	1D No-Ov	1D Ov
Core Local Store	$4N$	$6N$	$6N$	$8N$
Radix-4 Cycles	$6Nlog_4^N/n_r^2$			
Twiddle Mult Cycles	-	-	$6N/n_r^2$	$4N/n_r^2$
Communication	$4N = 2N$(R)+$2N$(W)		$6N = 4N$(R)+$2N$(W)	

Fig. 6. Different FFT core requirements for both overlapped and non-overlapped versions of $N \times N$ 2D and N^2 1D FFTs.

memory system could be designed to handle the required irregular addressing in either case, but the control is somewhat simpler if a transposition is included while reading any data for both sets of DFTs, rather than combining it with the bit-reversal permutation in the writeback of the results of the first set of DFTs.

The multiplication by the "twiddle factors" can easily be handled in either the transposed or natural ordering. In this design, it occurs after the data is read back into the core for the second set of transforms, but before the second set of transforms begins. The second DFT step therefore requires twice the read bandwidth of the first set of DFTs, since both the intermediate data and the twiddle factors must be read from the SRAM to the core for this step.

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

This completes 1D FFT, but the values are stored in the transpose of the natural order. Given the ability of the SRAM to source data in transposed order, it is assumed that subsequent computational steps will simply request the data in the right format.

2) Algorithm for 2D FFTs: For a core capable of computing 1D FFTs of lengths $64, \ldots, 4096$, two-dimensional FFTs of sizes up to 4096×4096 are straightforward. These transforms are similar to large 1D FFTs, but are simpler to implement since there are no additional "twiddle factors" required. Since the row transforms can be performed on naturally contiguous data before the column transforms, the 2D FFT also requires less data transposition. The data motion for the 2D FFT is shown in the right panel of Figure 5. After the row transforms are produced and written back to the SRAM (using bit-reversal to obtain natural ordering), the data is read in transposed order to allow computation of the column transforms. Once the second set of transforms are computed and written back to memory, the SRAM will hold the transpose of the full 2D FFT of the input data. As in the 1D case, subsequent processing steps can either use the data in place with its transposed ordering or read the data out in transposed order.

VI. Architecture Trade-offs and Configurations

In previous sections, we provided the fundamentals for mapping a Radix-4 based FFT transform to a modified LAC. In this section, we describe the necessary modifications to the PEs, the core, and the off-core SRAM to support the efficient mapping of FFTs. We first describe analytical models before demonstrating the tradeoff analysis using them.

A. Analytical models

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 III. Therefore, an N-point FFT requires a cycle count of $Total_{Cycles} = N/4 \times 24 \times log_4^N/n_r^2$.

We consider two cases in our analysis for FFT on the core: no or full overlap of communication with computation. Note that the FFT operation has a much higher ratio of communication/computation ($O(N)/O(N \log N)$) compared to a typical level-3 BLAS operation like matrix multiplication ($O(N^2)/O(N^3)$). Therefore, the non-overlap FFT solution suffers significantly resulting in low utilization. The different cases of the core requirements are presented in Figure 6.

a) 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 and columns. The twiddle factors remain the same and therefore the core bandwidth and local store size can be calculated as follows. The amount of data 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 $BW_{Avg} = 2n_r^2/3 \log_4^N$. 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 $BW_{eff} = 2n_r^2/3(\log_4^N - 1)$.

The aggregate local store of PEs includes the N complex input points and N complex twiddle factors. In the no-overlap case, this amount of storage suffices since 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 $6N$ floating-point values.

b) Core constraints for 1D FFTs: The second set of FFTs in the "four-step" 1D FFT, require more input bandwidth to the cores. Each core is performing independent FFT operations on rows. The twiddle factors are changing with each new N point input vector. However, each twiddle factor is going to be multiplied with the corresponding input before the FFT computation gets started. An extra $4N$ real multiplications are added to the total computations of this transform. Therefore the total cycle count is $Total_{Cycles} = (6Nlog_4^N + 4N)/n_r^2$. The amount of data transfer for a problem of size N includes $2N$ complex inputs (transform inputs and 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 can be derived from the division of the total data transfers by the total computation

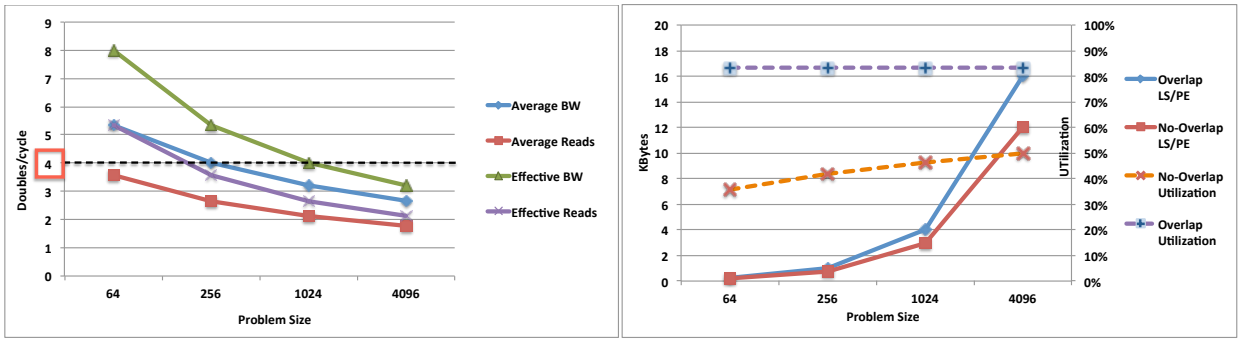

Fig. 7. Left: required bandwidth to support full overlap in the worst case for different problems. Note that four doubles/cycle is the maximum capacity of a core with column buses used for external transfers. Right: local store/PE and respective utilization for both cases of non-overlap and overlapped solutions.

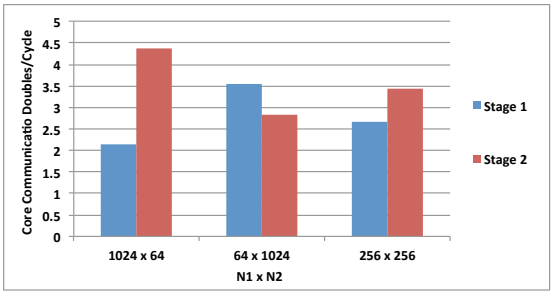

Fig. 8. Average communication load on core for 64K 1D FFT.

cycles as $BW_{Avg} = 3n_r^2/(3\log_4^N + 2)$. However, 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 $BW_{eff} = 3n_r^2/(3\log_4^N - 1)$ (see Figure 7).

Each N-point input to the core has to be pre-multiplied by a different set of twiddle factors, so another buffer is needed for the corresponding twiddle factors.

Finally, as described earlier, one can compute the 1D discrete Fourier transform 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 put 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. Figure 8 demonstrates this effect for the case of a 64K point 1D FFT with three different options for $64K = N_1 \times N_2$.

B. Architecture Configuration

In this section, we present a set of core modifications suggested by the preceding analyses. Several options will be presented for both the PE and the core to meet the data handling and access pattern demands of the problem.

a) Core Configuration: Figure 7 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 FMA 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 LAC. However, for the overlapped algorithm at problem sizes $N <= 1024$, extra off-core bandwidth is required to attain the peak achievable efficiency. The chart on the left side of Figure 7 shows that the maximum required off-core bandwidth does not exceed eight doubles/cycle. Therefore, the off-core bandwidth needs to double that of the original LAC design. Furthermore, the PE 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.

Doubling the memory bandwidth could be implemented in three ways: doubling the width of the column buses, doubling the number of column buses, or connecting the row buses to the off-core memory. The first choice would be complex to implement, since the original column bus bandwidth is matched to the PE-local SRAM bandwidth. The second choice is not quite as complex, but still requires an expansion of the PE local SRAM bandwidth. The best solution is therefore to expand the memory interface so that both row and column buses can transfer data to/from PEs. This solution does not impose any area overhead for additional broadcast buses and provides an interface to the memory that is always free of inter-PE use during phases in which the column buses are busy with inter-PE transfers. Further, this design is symmetric and natively supports transposition.

b) PE Configuration: The PE micro-architecture must perform the three tasks of Radix-4 butterfly computation, FFT communication, and off-core communication concurrently. Some extra logic and storage is needed to facilitate data movements and locality. These options are described with the help of Figure 9.

An 8-byte register file is needed to store the four complex input, temporary, and output values of the FMA-optimized Radix-4 butterfly (Figure 1). The twiddle factors take an extra four registers. We separate these two register files to avoid adding extra ports to the existing (large) register file and hence save energy and area. The PE SRAM needs enough bandwidth to provide data for both Radix-4 computations and off-core communications. Each butterfly has six complex inputs and produces four complex outputs. This data transfer would require 20 cycles from a typical single-ported 8-byte wide SRAM. The remaining four cycles of the 24-cycle radix-

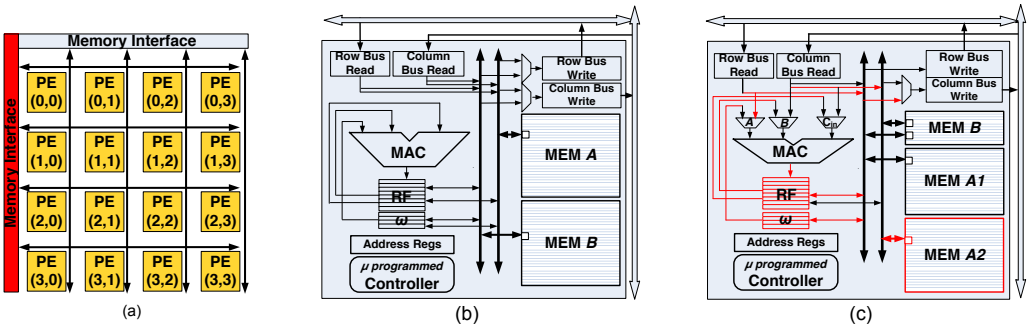

Fig. 9. New core and PE configurations for full-overlap FMA-optimized Radix-4 FFT: (a) Core with extended external row bus interface (b) FFT-optimized core with two 8-byte, single-ported SRAMs (c) Modified linear algebra PE with two 8-byte, single-ported SRAMs to contain matrix A.

16Kbyte SRAM	Wide	Dual-port	Separate
# SRAMs, # ports x bus-width	1,1x16	1, 2x16	2, 1x8
Cycle time (nS)	0.73	0.79	0.67
Energy per access (nJ)	0.010	0.009	0.005
Total area (mm^2)	0.054	0.141	0.054
Max Power at Target Freq (mW)	0.010	0.017	0.010
Worst case FFT Access/Cycle	0.613	1.227	1.227
Worst Case FFT total dynamic energy (J)	0.006	0.011	0.006

Fig. 10. PE SRAM options and their area, performance, and energy consumption report by CACTI [18].

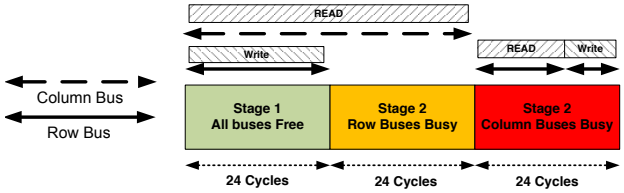

Fig. 11. Schematic of data bus usage for fully overlapped pre-fetch/post-store for the worst case of a 64-element FFT.

4 compute phase do not provide enough time to implement the required off-core communications. There are three solutions to provide the required bandwidth to the PE-local stores: an extra port to the same PE SRAM could be added, a wider (16 byte wide) port could replace the existing port, or a separate SRAM block with its own 8-byte port can be added.

A simple study of memory power and area consumption of these options is presented in Figure 10. The dual ported solution consumes much more power and area than the other two. Hence, the wide solution needs extra buffering and a more complicated control to transmit data to/from other components. The two SRAM solution is the best one with the simplest control. This FFT PE is presented in Figure 9(b). It has a symmetric design with two separate buses – each is connected to all the components in the PE and to one of the SRAMs.

So far, we have described the options for an FFT PE that is based on the baseline architecture but is specifically designed for FFT operation. If one starts with an existing linear algebra PE to make a hybrid FFT/Linear Algebra architecture, the register file design has to be extended with more ports and more capacity to match the requirements of the FFT. There are two options for extending this micro-architecture to facilitate FFT bandwidth for the hybrid design. The original linear algebra PE has one larger, single-ported SRAM and one smaller, dual-ported SRAM. Since the smaller SRAM is already dual ported, we must modify the larger SRAM to provide extra bandwidth. As discussed above, the best solution is to divide the larger SRAM into two halves and adding an extra bus to the PE (see Figure 9(c)).

c) Off-core Memory Configuration: As noted in Section VI, the maximum core bandwidth required for the non-overlapping case is four double-precision elements per cycle. The non-overlapped configuration requires an effective bandwidth of up to eight double-precision elements per cycle for problems sizes smaller than N=1024. Core changes are required to support external bandwidths above four double-precision values per cycle, with the addition of memory interfaces on the row buses providing the most symmetric solution. The effective bandwidth required for pre-fetch/post-store is decreased by opening up more cycles in which at least one of the buses is not used.

For the case of double-precision complex data, the natural data size is $2 \times 64 = 128$ bits. We consider whether a memory with four ports (2 read, 2 write), each with 128 bit width, can be matched to the instantaneous read and write requirements of the core.

The worst case occurs for $N = 64$, 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 72 cycles required to perform the three radix-4 computations. The proposed memory interface bandwidth is clearly adequate to provide for the average traffic – external SRAM requires 64 cycles (of the 72 available) to provide the 256 words prefetched by the core for its next iteration. The writes require only 32 of the 72 cycles, and these can be overlapped with the reads.

The detailed scheduling is not particularly complex, but does require careful design, as shown in Figure 11. Recall (Figure 3) that during the first radix-4 step (24 cycles) of the 64-point FFT neither row nor column buses are in use, while the row buses are in use during the second (24-cycle) radix-4 step and the column buses are in use during the third 24-cycle radix-4 step. Since the SRAM requires 64 cycles to source the read data, it is clear that reads must occur during all three of these phases, and that reads must occur on the column buses during phase 2 (while the row buses are busy) and on the row buses during phase 3 (while the column buses are busy). Similarly, the writes require 32 cycles at the SRAM, so they must occur during at least two of the three phases.

If we further assume that only a single SRAM bank within

each PE is available for this pre-fetch/post-store communication (with the other bank being used for the concurrent computation step), then a PE can read or write to a row or column bus, but cannot use both row and column buses in the same cycle without additional buffering. Fortunately, due to the shared bus architecture, each PE can only write to the column bus on 1/4 of the cycles and can only write to the row bus on 1/4 of the cycles, so it is straightforward to swizzle the active PEs so that no PE is both reading and writing on the same cycle. 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.

VII. Experimental Results and Implementations

In this section, we present area, power and performance estimates for the LAC with the modifications introduced in previous sections.

A. Area and Power Estimation

The basic PE and core-level estimations of a LAC in 45nm bulk CMOS technology are reported in [3], [5]. There, we show that operation at 1GHz provides the best tradeoff between performance and efficiency. Power and area of floating-point units use the measurements reported in [19]. CACTI [18] is used to estimate the power and area consumption of memories, register files, look-up tables and buses.

Figure 12 reports the projected power and area consumption of the components of the PE for the three different designs, along with the corresponding design metrics. The power consumption of the FFT design is considered for the worst case and highest possible number of accesses. For the hybrid design, we report a pair of numbers, one for GEMM and one for FFT.

Figure 13 summarizes the power breakdown of the three proposed designs. For the pure FFT and hybrid cores, the "actual" power considers the worst case power consumption when running an FFT. The maximum power breakdown shows the "maximum power" that is used by the three different PE designs. We observe that the power consumption is dominated by the FPMAC unit, with secondary contributions from the PE-local SRAMs.

Figure 14 demonstrates the efficiency metrics of the three different PE designs. We can observe that the hybrid design has lower efficiency when considering maximum power and area. However, since the leakage power consumption of the SRAM blocks are negligible, the actual power efficiency is maintained in the hybrid PE. Note that in all cases, the efficiency numbers are already scaled by achievable utilization of in all cases. The area breakdown emphasizes that most of the PE area is occupied by the memory blocks. The hybrid design has the largest aggregate PE SRAM capacity.

B. Comparison to Other Processors

Figure 15 provides comparisons of estimated performance, area, and power consumption between our proposed design and several alternative processors [20], [21], [22], [23]. In each case, we limit the comparison to double-precision 1D FFT performance for problem sizes that fit into either the first and/or second levels of SRAM or cache. All area and power estimates are scaled to 45nm technology and include only the cores and SRAM. Such comparisons are necessarily coarse due to the disparate nature of the data sources – some are based on detailed engineering specifications, some on measurements with full-system hardware, and in a few cases power consumption values were estimated based on published thermal design power of products (and then adjusted as discussed above). In each case the proposed FFT engine provides at least an order of magnitude advantage in performance per watt and unit area.

PE Design	LAC	FFT	Hybrid
SRAM			
Total SRAM Area (mm^2)	0.070	0.054	0.073
Total SRAMs MAX Power (W)	0.013	0.010	0.015
Total SRAM Actual Dynamic Power (W)	0.005	0.006	0.006
Floating-Point Unit			
FP Area (mm^2)	0.042	0.042	0.042
FP Power (W)	0.031	0.031	0.031
Register File			
RF Area (mm^2)	0.000	0.008	0.008
RF MAx Power (W)	0.000	0.004	0.004
RF Actual Power (W)	0.000	0.003	0.003
Broad-cast Buses			
Bus Area /PE (mm^2)	0.014	0.014	0.014
Max Bus Power (W)	0.001	0.001	0.001
PE Total			
Total PE Area (mm^2)	0.126	0.119	0.138
Total PE MAx Power (W)	0.045	0.047	0.052
Total PE Real Power (W) (GEMM,FFT)	0.037	0.041	(0.037, 0.041)
GFLOPS/W (GEMM, FFT)	53.82	40.53	(53.80, 40.50)
GFLOPS/MAX W (GEMM, FFT)	44.59	35.80	(38.55, 32.12)
GFLOPS/mm^2 (GEMM, FFT)	15.84	14.00	(14.54, 12.11)
W/mm^2	0.334	0.391	0.377

Fig. 12. PE designs for dedicated LAC, dedicated FFT, and a hybrid design that can perform both operations.

VIII. Conclusions and Future Work

Starting with a baseline linear algebra architecture, this paper presents analysis and modification of the design to efficiently support 1D and 2D complex FFT algorithms. A thorough analysis of the similarities and differences between the BLAS3 and FFT algorithms at the level of computational data dependence, inter-PE communication, and off-core communication was performed. We demonstrate how careful algorithm analysis for the target architecture, combined with judiciously chosen data-path modifications, allowed us to produce a highly efficient accelerator for FFT operations with minimal changes to the original linear algebra core. Finally, we present a hybrid core that can perform both algorithms while maintaining the efficiency characteristic of the original application-specific design.

Our results show that this hybrid design can achieve up to 40 GFLOPS/W power efficiency for double-precision complex FFTs with 83% effective utilization of the FMAC units. For future work we plan to investigate multi-core versions of our design, including exploration of the next layers of memory hierarchy down to DRAM. We also plan to look into further specialization of FFT designs on the same core to achieve better utilization and efficiency using custom built FP units.

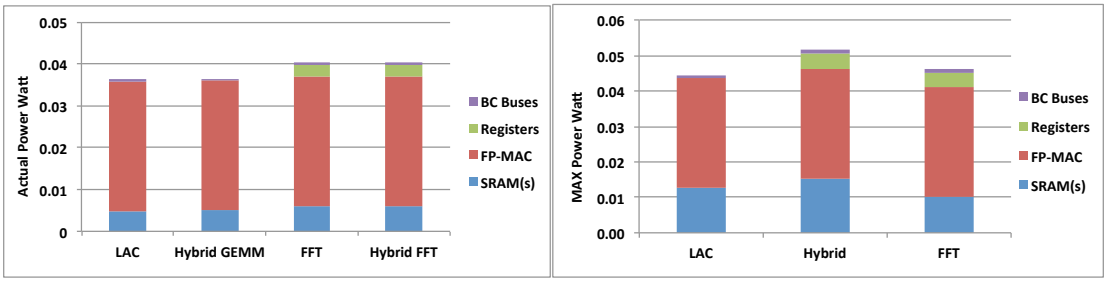

Fig. 13. Actual and maximum PE power consumption of each design for target applications at 1GHz.

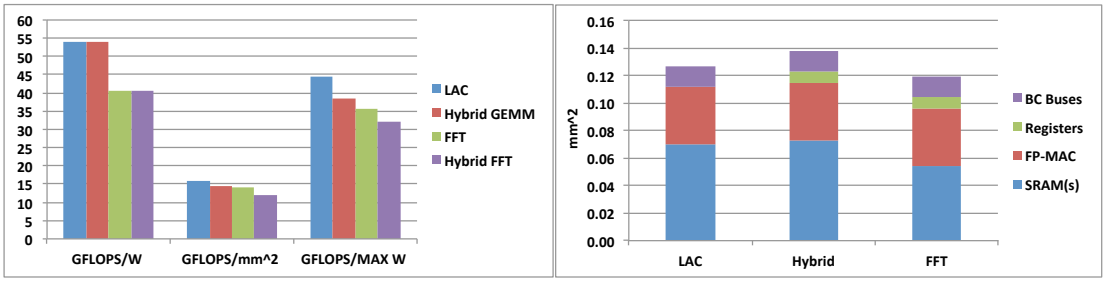

Fig. 14. Left: Efficiency parameters for the target applications. Right: Total area breakdown of the PE for each design.

Platform Running FFT	Problem size fits in	Cache/SRAM KBytes	Peak GFLOPS	FFT Nominal GFLOPS	Power (Watt)	Area (mm^2)	GFLOPS/Watt	GFLOPS /mm^2	Utilization
Hybrid Core	On-core SRAM	288	32.0	26.7	0.66	2.2	40.50	12.12	83.3%
Hybrid Core+ SRAM	Off-core SRAM	2336	32.0	26.7	0.96	13.2	27.7	2.01	83.3%
Xeon E3-1270 (1 core)	L1 cache	32	27.2	18.0	28	27.7	0.64	0.65	66.2%
ARM Cortex A9 (1 GHz)	L1 cache	32	1.0	0.6	0.28	6.3	2.13	0.09	60.0%
PowerXCell 8i SPE	SPE local RAM	2048	102.4	12.0	64	102	0.19	0.12	11.7%
NVIDIA Tesla C2050	L1+L2 cache	1728	515.2	110.0	150.00	529.0	0.73	0.21	21.3%

Fig. 15. Comparison between the proposed hybrid core and several alternatives for cache-contained double-precision FFTs scaled to 45nm.

ACKNOWLEDGMENTS

This research was partially sponsored by NSF grants CCF-1218483. 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).

REFERENCES

[1] M. Cheney *et al.*, *Fundamentals of radar imaging*, ser. CBMS-NSF regional conference series in applied mathematics. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2009.

[2] A. Kak *et al.*, *Principles of computerized tomographic imaging*, ser. Classics In Applied Mathematics. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2001.

[3] A. Pedram *et al.*, "A high-performance, low-power linear algebra core," in *ASAP*. IEEE, 2011.

[4] P. Milder *et al.*, "Computer generation of hardware for linear digital signal processing transforms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, no. 2, 2012.

[5] A. Pedram *et al.*, "Co-design tradeoffs for high-performance, low-power linear algebra architectures," *IEEE Trans. on Computers*, 2012.

[6] ——, "A linear algebra core design for efficient Level-3 BLAS," in *ASAP*. IEEE, 2012.

[7] ——, "Floating point architecture extensions for optimized matrix factorization," in *ARITH*. IEEE, 2013.

[8] G. Bergland, "Fast Fourier transform hardware implementations–an overview," *Audio and Electroacoustics, IEEE Transactions on*, vol. 17, no. 2, pp. 104 – 108, jun 1969.

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

[10] E. S. Chung *et al.*, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?" *MICRO '43*, pp. 225–236, 2010.

[11] B. Akin *et al.*, "Memory bandwidth efficient two-dimensional fast Fourier transform algorithm and implementation for large problem sizes," in *FCCM 2012*. IEEE, 2012, pp. 188–191.

[12] K. S. Hemmert *et al.*, "An analysis of the double-precision floating-point FFT on FPGAs," in *FCCM '05*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 171–180. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2005.16

[13] H. Karner *et al.*, "Top speed FFTs for FMA architectures," 1998.

[14] F. G. Van Zee and others. FLAME Working Note #66, "BLIS: A framework for generating blas-like libraries," The University of Texas at Austin, Department of Computer Sciences, Technical Report TR-12-30, November 2012.

[15] S. Jain et al., "A 90mW/GFlop 3.4GHz reconfigurable fused/continuous multiply-accumulator for floating-point and integer operands in 65nm," *VLSID '10.*, 2010.

[16] A. Pedram *et al.*, "On the efficiency of register file versus broadcast interconnect for collective communications in data-parallel hardware accelerators," *SBAC-PAD*, 2012.

[17] D. H. Bailey, "Ffts in external of hierarchical memory," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. ACM, 1989, pp. 234–242.

[18] N. Muralimanohar *et al.*, "Architecting efficient interconnects for large caches with CACTI 6.0," *IEEE Micro*, vol. 28, 2008.

[19] S. Galal et al, "Energy-efficient floating point unit design," *IEEE Trans. on Computers*, vol. PP, no. 99, 2010.

[20] M. Yuffe *et al.*, "A fully integrated multi-CPU, GPU and memory controller 32nm processor," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*. IEEE, 2011.

[21] J. Demmel *et al.*, "Instrumenting linear algebra energy consumption via on-chip energy counters," UC at Berkeley, Tech. Rep. UCB/EECS-2012-168, 2012.

[22] M. Kistler *et al.*, "Petascale computing with accelerators," in *PPOPP 2009*. ACM, 2009, pp. 241–250.

[23] D. Wu *et al.*, "Implementation and evaluation of parallel fft on engineering and scientific computation accelerator (esca) architecture," *Journal of Zhejiang University-Science C*, vol. 12, no. 12, 2011.