An Analysis of the Double-Precision Floating-Point FFT on FPGAs

K. Scott Hemmert Keith D. Underwood

Sandia National Laboratories*
P.O. Box 5800, MS-1110, Albuquerque, NM 87185-1110
E-mail: {kshemme,kdunder}@sandia.gov

Abstract

Advances in FPGA technology have led to dramatic improvements in double precision floating-point performance. Modern FPGAs boast several GigaFLOPs of raw computing power. Unfortunately, this computing power is distributed across 30 floating-point units with over 10 cycles of latency each. The user must find two orders of magnitude more parallelism than is typically exploited in a single microprocessor; thus, it is not clear that the computational power of FPGAs can be exploited across a wide range of algorithms. This paper explores three implementation alternatives for the Fast Fourier Transform (FFT) on FPGAs. The algorithms are compared in terms of sustained performance and memory requirements for various FFT sizes and FPGA sizes. The results indicate that FPGAs are competitive with microprocessors in terms of performance and that the "correct" FFT implementation varies based on the size of the transform and the size of the FPGA.

KEYWORDS: IEEE floating point, FFT, Fast Fourier Transform, FPGA, reconfigurable computing

1. Introduction

Field-programmable gate arrays (FPGAs) have long been attractive for accelerating fixed-point applications. Early on, FPGAs could deliver tens of narrow, low latency fixed-point operations. Designers only needed to find a limited amount of parallelism and the most popular applications (image processing) had inherent parallelism. As FPGAs matured, the amount of parallelism to be exploited grew rapidly with FPGA size. This was a boon to many application designers as it enabled them to capture more of the application. It also meant that the performance of FPGAs was growing faster than that of CPUs[20]. Thus, it was possible to waste some of peak performance and still dramatically outperform CPUs.

The design of floating-point applications for FPGAs is much different. Whereas FPGAs have long had a significant advantage over CPUs in raw integer performance, they are just now reaching parity with CPUs in raw floating-point performance. While FPGAs may be able to sustain a higher percentage of their peak floating-point performance than CPUs, they face some challenges in that quest. The foremost issue for FPGAs is the need to extract massive amounts of parallelism. This is clearly possible for matrix multiplication[21, 22], LU decomposition[8], and matrix vector multiplication for both dense[21] and sparse[23, 5] matrices, but it can be difficult in other cases. With 30 floating-point units having 10 cycles of latency, it may be impossible to find the 300 way parallelism that is implied while satisfying all of the inherent data dependencies.

The fast fourier transform (FFT) is a next step in exploring the floating-point capabilities of FPGAs. While it has abundant parallelism, it introduces more data dependencies. While the FFT has been studied on FPGAs before, it has not been considered using double precision floating-point, which is critical to scientific applications. This changes the issues significantly because addition becomes just as expensive as multiplication.

This paper presents implementation options for the complex, double precision floating-point FFT (radix-2). While radix-2 implementations are not general (mixed radix support will be required), they serve to illustrate most of the important points. Three implementations are considered that have significantly different characteristics. They are compared (with each other and to microprocessors) based on sustained FLOPs, internal memory requirements, and external memory bandwidth requirements relative to the size of the FFT. The results indicate two things: FPGAs are competitive in performance with microprocessors, and the "right" implementation depends on the size of the FFT being performed as well as the size of the FPGA being used.

Previous efforts studying FFTs and FPGAs as well as other floating-point algorithms are presented in Section 2.

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

The fast fourier transform algorithm is discussed in Section 3 followed by discussions of the three architectures examined in Sections 4, 5, and 6. The three architectures are compared in Section 7. Conclusions are presented in Section 8 followed by future work in Section 9.

2. Related Work

Signal processing is a long time application for FPGAs [4]. Some of the earliest floating-point work considered FFTs [17], but found IEEE floating-point arithmetic to be impractical on FPGAs. Further work on fixed-point FFTs applied CORDIC arithmetic to the problem [6]. Designers have continued to tweak the implementations of the FFT on reconfigurable computing platforms [15].

Recent research has again focused on the FFT on FPGAs. In [10], the FFT was mapped to the MorphoSys reconfigurable computing platform to achieve higher performance than other advanced architectures. Others have transformed the architecture of the FFT implementation to reduce power [3] to meet the growing demand for low-power signal processing. Finally, [19] presents an implementation of an alternative radix-2 FFT derivation using Handel-C.

This work is most similar to [3] in that it explores the issues of both horizontal and vertical parallelism; however, the move from 16 bit data to double precision floating-point significantly changes the design space. Floating-point arithmetic requires much more area per operation and, more importantly, it requires almost as much area for an adder as a multiplier. Furthermore, it has much higher demands on memory capacity and bandwidth. This work further differs from [3] in that it explores the performance implications as various architectures scale to future generations of FPGAs.

A number of efforts have also begun to consider floating-point on FPGAs. Notably, a number of vendors sell a single precision floating-point FFT core [2, 1, 14], but they generally do not discuss the details of their implementations or why various design choices were made. Furthermore, they use the single precision format, which is currently more practical on FPGAs. Numerous efforts have also begun to study double precision floating-point from the relatively simple matrix multiply[22] and BLAS[21] operations to the more complex LU decomposition[8] and sparse matrix-vector operations[23, 5]. This effort continues the exploration of the rapidly increasing double precision floating-point capability of FPGAs[20].

3. The Fast Fourier Transform

The FFT is an optimized implementation of the Discrete Fourier Transform (DFT). The DFT is a traditional signal processing technique that is applied to a variety of domains from speech processing to sonar beam forming[9]. In many domains, fixed-point arithmetic is sufficient; however, the DFT also has uses in scientific applications ranging from global climate modeling [16] to molecular dynamics[12, 11, 13] that require double precision floating-point arithmetic. The fundamental calculation of the N point DFT is described as:

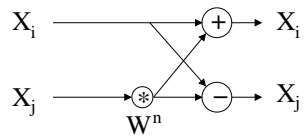


Figure 1. The basic butterfly operation

$$Y[j] = \sum_{k=0}^{N_1-1} X[k]W_N^{jk}; \ \texttt{where} \ W_N^{jk} = e^{\frac{-i2\pi jk}{N}} \quad (1)$$

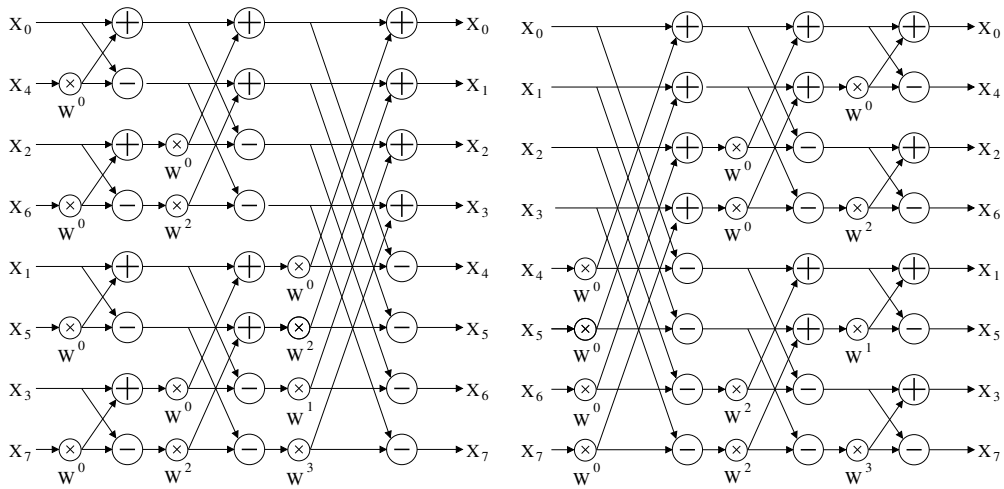
When implementing the DFT as an FFT[18], there are many ways to structure the computation. The first decision to make — the radix of the operation — determines how many data items are combined in any given stage. Radix-2, where each stage of computation operates pairwise on the data set, was chosen for several reasons, including its simplicity. Radix-2 also yields the smallest butterfly unit, which allows for greater flexibility in studying the design space. Other radices reduce the total number of operations, but increase the complexity and reduce the flexibility. Similarly, multiple-radix designs can increase flexibility.

Given a radix, there is still flexibility in formulating the computation. One formulation yields a basic computational kernel consisting of a single complex multiplication and two complex additions, as shown in Figure 1. This structure is commonly referred to as a butterfly because of the crossing dependencies. The radix-2 FFT is made up of $log_2(N)$ stages, where each stage computes $\frac{N}{2}$ butterflies.

There are two main ways to structure the stages, shown in Figure 2. A butterfly unit in a stage operates on a pair of results from the previous stage separated by an increasing (a) or decreasing (b) distance. Each of these structures requires a data reording, either on the front- or back-end. We chose to reorder on the back-end as it allowed us to do the biggest butterflies first, providing more independent data sets with each progressive stage. Both approaches produce the same results, but in a different order.

The butterfly computation requires 4 real multiplications and 6 real additions. The hardware used in this study does these computations using 2 floating-point multipliers and 3 floating-point adders, as shown in Figure 3[1]. On average,

[1] S is a switch that directs inputs to alternate outputs. R's replicate the input once. C is a crossover to facilitate the complex multiply.



(a) Reordered inputs **(b)** Reordered outputs

Figure 2. Two variations of 8-point, radix-2 FFTs

the hardware accepts one new complex number per cycle and generates one complex output per cycle. Each floating-point unit is used twice for each set of inputs, providing the total operations required for the butterfly. This design for the butterfly unit was chosen because it provides the greatest flexibility. It only requires a memory bandwidth of one complex data item per cycle, letting memory bandwidth scale more evenly. The bandwidth in and out of the unit also matches the bandwidth of on-chip dual-port block rams, allowing a single butterfly unit to be coupled with a single set of block rams on the chip.

In evaluating the performance of the FFT, the floating-point operation count that is typically used is $5Nlog_2(N)$; there are $log_2(N)$ stages that each contain $5N$ computations (four multiplies and six additions for each pair of data). These numbers only hold strictly for radix-2, though they are a good approximation for other radices. As the radix increases, the number of stages goes down, but stage complexity increases.

The minimal data transfers to memory for the FFT is $2N$ elements with each element being 16 bytes for double precision complex numbers. This gives $32N$ bytes for a best case overall bytes per FLOP requirement of $\frac{32}{5log_2(N)}$. The actual bytes per FLOP requirement, however, will depend on how much of the data can be successfully cached on chip. This is discussed further in the discussion of each algorithm.

4. Parallel FFT Implementation

Parallelism in the FFT computation can be exploited in two ways: pipelined units, or parallelism in the stages (S), and parallel units, or parallelism (P) within a stage. The parallel design point explores the extreme where all of the

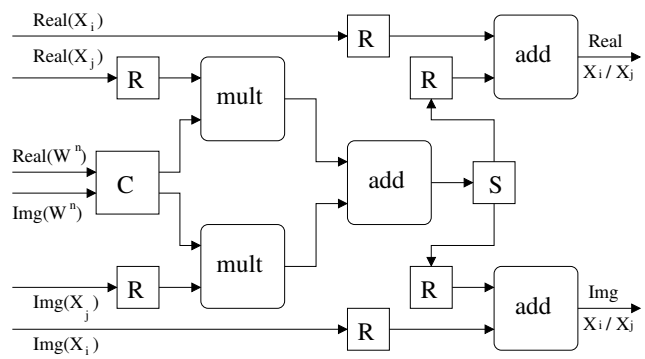


Figure 3. Basic butterfly datapath.

parallelism is within a single stage as shown in Figure 4(a). In this mode, data is read from external memory, processed iteratively, and written to external memory. Each of the butterfly units operates on a different range of the data; each unit iterates (mostly) independently through all the stages of the computation. All of the butterfly units are used for the entire computation, but the overall throughput is constrained by external memory bandwidth. The number of cycles needed to compute an FFT using this scheme is:

$$T = \frac{2N}{BW} + BL + (\frac{N}{P} + BL)(log_2(N) - 2) \quad (2)$$

The first term of Equation 2 is the time to read and then write N items based on the memory bandwidth, BW, in terms of complex double precision floating-point items per cycle. The usable bandwidth is limited to the number of units, P. The second term is the latency of passing through

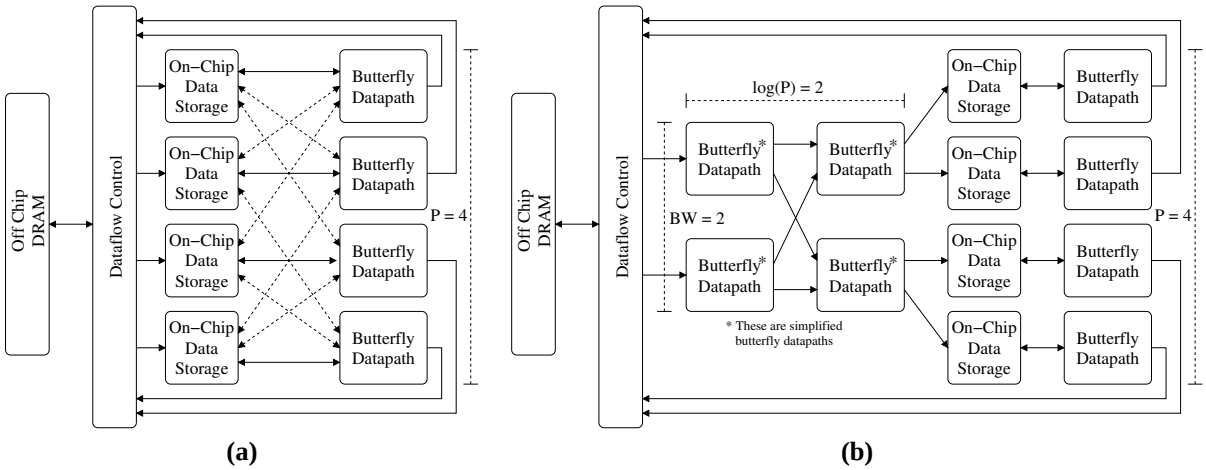


Figure 4. (a) Exploiting higher parallelism (P); **(b)** An optimization to leverage extra FPGA area

the butterfly units during the read from memory. The third term is the time to perform the iterations — using P butterfly units of latency BL for $log_2(N) - 2$ iterations assuming that the first and last iteration are performed as part of reading and writing the data.

This design has two significant limitations. First, there are $log_2(P)$ stages in which the units must communicate, where P is the number of parallel units. This arises from the largest butterflies, which cannot have all of the data held in an individual RAM. Second, the number of butterfly units is a power of two (the radix), which can prevent the implementation from using the entire FPGA. Both of these issues can be improved through the use of "lead-in" units. By using $log_2(P)$ lead-in stages to compute the largest butterflies, the data becomes P way independent, as shown in Figure 4(b). The parallelism in the lead-in stages is determined by the available memory bandwidth. Each butterfly unit is able to handle one new complex data item per clock cycle.

The first two of these lead-in stages can be optimized. The first stage uses only one constant, $1 + 0i$, and the second stage uses only $1 + 0i$ and $0 + 1i$. Since these constants do not require a multiply, the circuitry in the butterfly units in the first two stages is reduced to two floating-point adders. Using these lead-in stages reduces the number of iterations that the parallel portion of the circuit must perform at the penalty of adding a startup latency to get through these units. The new cycle count for completion becomes:

$$T = \frac{2N}{BW} + BL + Startup + (\frac{N}{P} + BL)(log_2(N) - (2 + log_2(P))) \quad (3)$$

All of the FFT designs require internal storage for both data and "twiddle factors" (W_N^{jk}). In the parallel implementation, the data storage requires N double precision complex data items. This one storage area is reused by all iterations and must supply sufficient bandwidth to support the number of parallel processing units; thus, using off-chip memory is not practical. The total number of twiddle factor constants needed is equivalent to $\frac{N}{4}$ storage locations of 16 bytes (complex double-precision numbers). The prototype implementation has four parallel units.

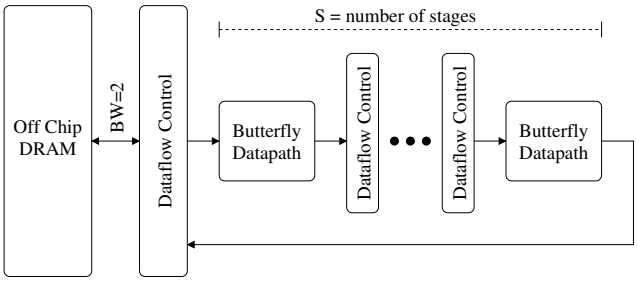


Figure 5. A pipelined FFT implementation

5. Pipelined FFT Implementation

At the other extreme, one butterfly unit can be dedicated to each of the stages of the FFT in a pipelined fashion as is illustrated in Figure 5. Data is read from memory and passed through a series of butterfly units before being written back to memory. Data delays and permutations are needed between each of the stages and between the pipelined FFT unit and DRAM memory. When the number of stages, S, that can be implemented in the FPGA is less than the number of stages needed by the FFT ($log_2(N)$), then $\frac{log_2(N)}{S}$ passes to memory are needed, with a the final pass being a subset, R, of the stages. For each pass to

memory, data must be read and written in a particular permutation to optimize the delay and storage requirements in the pipeline (not described here). The number of cycles required to compute an FFT using a pipelined approach is:

$$T = P(S) \times \left\lfloor \frac{log_2(N)}{S} \right\rfloor + P(R) \quad (4)$$

$$P(J) = BL \times J + I(J) + \frac{2N}{BW} + (B-1) \times 2^J \quad (5)$$

$$I(K) = \sum_{i=0}^{K-1} B \times 2^i \approx B \times 2^K \quad (6)$$

$$R = log_2(N) \ \texttt{mod} \ S \quad (7)$$

Each pass, $P(J)$, through J butterfly stages (each having a latency of BL) requires the time shown in Equation 5. Data dependencies between the stages introduce a delay that doubles at each stage and create a total inter-stage delay given by $I(K)$. The burst length of standard DRAM memories introduce a penalty associated with the burst length, B, to both the interstage delay and a back-end reordering time. The time to retrieve the data from memory and write it back is defined by $\frac{2N}{BW}$; however, a note is in order. The usable bandwidth is limited to one complex double precision floating-point number per cycle per direction due to the limited throughput of the butterfly unit. The final term represents the final pass through a subset of the stages, R, with the corresponding delays.

The pipelined FFT implementation has two significant disadvantages. Foremost, the high latency of the overall pipeline means that many of the butterfly units sit idle while the pipeline is initialized and while it is flushed; thus, if the overall computation is short relative to the pipeline delay, the sustained performance is low. In addition, if the number of FFT stages is not an even multiple of the number of hardware stages, many of the hardware resources sit idle on the final pass. The primary advantages of this implementation are the limited requirement for memory bandwidth and the limited requirement for internal memory for relatively short pipelines. In the prototype implementation, for example, a six stage pipeline is implemented, which more fully utilizes the Virtex-2Pro 100 and only requires a fraction of the memory needed by the parallel implementation.

The on-chip memory requirements for the pipelined version are almost entirely dependent on the number of stages. The reordering buffer between butterfly stages is $\frac{3 \times 2^i}{2}$ elements, where i is the stage number. This gives a total of $\frac{3 \times 2^S}{2}$ items of storage for the items being transformed. Due to the organization of the computation, the twiddle factors require $\frac{3N}{8}$ locations to store the values in a way that provides the bandwidth to provide the twiddle factors to the appropriate butterfly units.

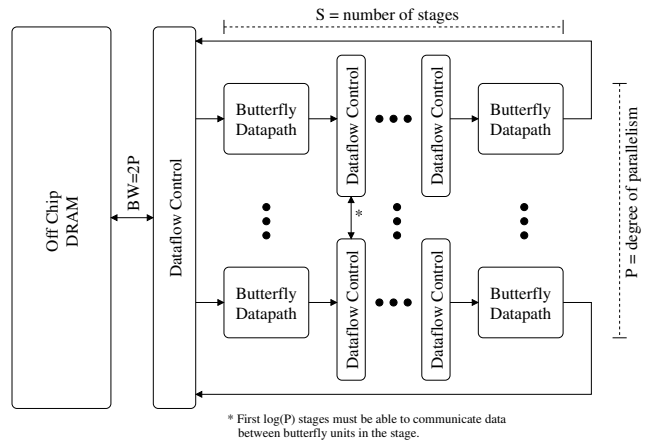


Figure 6. Parallel-Pipelined FFT Implementation

6. Parallel-Pipelined FFT Implementation

Figure 6 is the cross between the two previous architectures. Data moves from external memory, through a set of parallel pipelines, and back to external memory. The first $log_2(P)$ stages must have additional data exchange circuits (for the first pass through the pipeline) as these stages have data dependencies between the pipelines. This approach leverages the ability of the pipelined architecture to reduce bandwidth demands and the ability of the parallel architecture to tolerate shorter input vectors (as well as a wider variety of vector lengths) than the pure pipelined approach. In contrast, the parallel-pipelined hybrid has a higher bandwidth demand than the purely pipelined approach and less tolerance of short vectors than the parallel approach.

The number of cycles to compute the FFT using this approach is the same as that for the pipelined approach. The difference is that it raises the amount of bandwidth that the pipeline can accept and shortens the number of pipeline stages. The overall memory requirements, however, are significantly increased in the near term. The data storage requirements fall to $P \times \frac{3 \times 2^S}{2}$. Since the number of stages has been reduced by a factor of P, this is actually a factor of $2^P/P$ fewer iterms. For the twiddle factors, the storage grows to an upper limit of N storage locations (independent of the number of stages or the parallelism) to provide both the necessary storage and the necessary bandwidth.

7. Comparison of Architectures

The "right" FFT architecture varies not just with the size of the FFT, but also with the size of the FPGA. This section explores the performance of each of the architectures in two contexts. The first is the largest, fastest FPGAs available

today. The second is the next two generations of FPGAs, assuming a doubling of resources at each generation.

7.1. Performance and Area on Current FPGAs

The comparison of the architectures is based on the average sustained floating point operations per cycle (Figure 7(a)) given a number of butterfly units and an input bandwidth in complex, double precision floating-point items per cycle. In Figure 7, this comparison is in the context of the largest FPGAs available today — the Xilinx Virtex-2Pro-100. In this device, the parallel implementation can be configured with up to four parallel butterfly units (8 will not fit). Four "lead-in" units (a two wide lead-in path) will also easily fit, enabling the FFT implementation to accept two complex, double precision items per cycle (Parallel BW-2 Units-4). This equates to a 256 bit path to a relatively slow memory or a 128 bit path to a memory that is twice as fast as the floating-point units (both are reasonably achievable in current FPGAs and with a prototype currently in development). A four-wide lead-in path (8 lead-in units) would almost fit and would enable the design to leverage twice the memory bandwidth, although such a memory configuration would be expensive.

In the same FPGA, it is possible to fit 6 full butterfly units for either of the pipelined architectures. Note that the basic pipelined design can only use an external memory bandwidth of two complex, double precision floating-point items per cycle, but at larger matrix sizes, it achieves better performance than the parallel implementation that has more bandwidth. The pipelined approach shows its limitations both in the smaller FFT sizes, where the parallel implementation clearly wins, and in the saw tooth shape of the graph. This saw tooth shape arises from FFT sizes that require a number of stages that is not a multiple of S.

The parallel-pipelined architecture shown here uses a configuration with $P = 2$ and $S = 3$. If it is possible to provide the full usable bandwidth of four items per cycle, the parallel-pipelined approach clearly wins when N is greater than 1024. It has equivalent peak performance to the pipelined case, easily fits in the FPGA (unlike the parallel implementation for $BW = 4$), and has a shorter path than the purely pipelined approach, which limits the dips in the sawtooth. In contrast, when the full bandwidth cannot be provided, the parallel-pipelined approach is the clear loser as one of the two paths sits idle.

Another important point of comparison is the amount of memory required (Figure 7(b)). All three architectures have easily achievable memory requirements for FFTs of under 32K items, and the pipelined version can handle 128K. Since the parallel approach must store the entire vector being transformed internally, it has a significantly higher memory requirement than the pipelined approach. The parallel-pipelined approach has a similar memory demand to the parallel approach, but for a different reason: it must replicate the twiddle factors to provide sufficient bandwidth to the computational units.

7.2. Scalability on Future FPGAs

FPGAs are rapidly growing in density and special features that reduce area requirements for floating-point units. Thus, it is important to consider the impact of scale on algorithm performance. Figure 8 compares the performance of the various architectures for the next two doublings in FPGA size. It is assumed that memory bandwidth will double in the same time period, which is reasonable given the future of memory technology (DDR-2, FB-DIMMs).

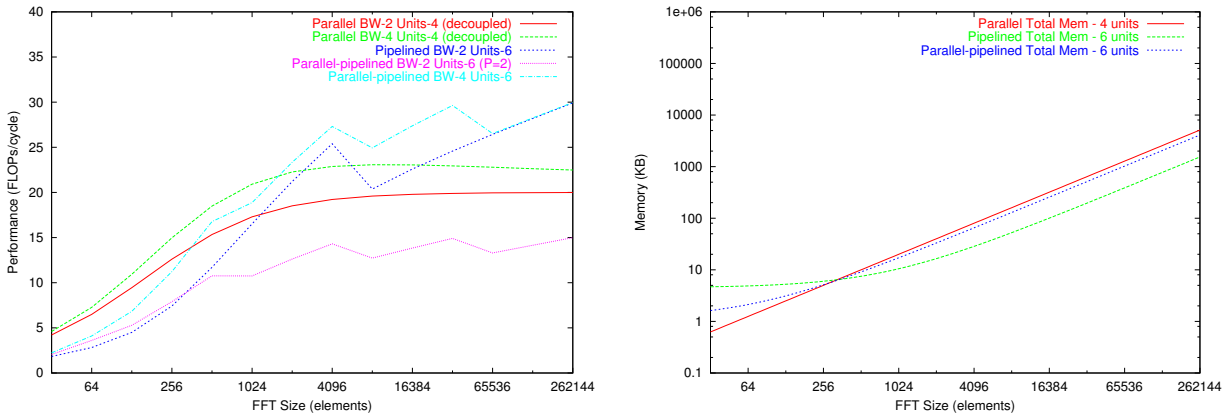
Figures 8(a) and (b) display the implications of doubling the FPGA capacity. Both the number of units and the memory bandwidth, have been doubled from Figure 7. In addition, to further explore the space, we consider the case of "slightly more than double" for the two pipelined paths[2]. It is important to note that the parallel path is no longer able to maintain the full set of $log_2(P)$ lead-in units needed to obtain full P-way decoupling, but the first two stages of simplified units can be retained.

The most notable result is that the extreme depth of the pipeline combined with the inability to leverage additional memory bandwidth leave the pipelined implementation at a clear disadvantage to the parallel implementation until nearly 256K samples are processed. The only configurations presented for the parallel-pipelined implementation match the memory bandwidth to the number of parallel pipes, P. The parallel-pipelined implementations shown have a larger advantage over the parallel approach than before as the depth of each parallel pipe has increased. Finally, Table 1 contains the parallelism, P, and stages, S, for the parallel-pipelined implementations. Correlating Figures 8 with Table 1 indicates that arrangements which are more "square" have better performance.

In terms of memory requirements, assuming that internal FPGA memory also doubles at each generation, the maximum size of an FFT that can be processed goes to approximately 128K points. This levels out because more and more internal memory is required to provide the bandwidth needed for all of the butterfly units. FPGA memories tend to run at the same frequency as the logic; thus, more ports are needed to feed more units. This is similar to, but different from microprocessors where the number of units is small and, therefore, so is the number of ports from cache.

Moving out one more generation yields 16 units for the parallel path that are not fully decoupled and 24 units for the pipelined approaches. The results for these are shown in Figures 8(c) and (d). As can be seen from these figures, the

[2]The parallel design must more than double to achieve BW-8 Units-8.



(a) **(b)**

Figure 7. A comparison of the **(a)** performance and **(b)**memory requirement of FFT implementations

Table 1. Parallel pipeline shapes

BW	Units	P	S	BW	Units	P	S
4	12	2	6	4	16	2	8
4	24	2	12	4	32	2	16
8	12	4	3	8	16	4	4
8	24	4	6	8	32	4	8
16	24	8	3	16	32	8	4

Table 2. Memory requirements summary

Arch	Min Memory	Max BW
Parallel	$N + \frac{N}{4}$	P
Pipelined	$\frac{3\times 2^S}{2} + \frac{3N}{8} + (B-1)\times 2^S$	2
Hybrid	$P \times \frac{3\times 2^S}{2} + N + (B-1)\times 2^S$	2P

purely pipelined approach no longer makes sense in terms of performance for reasonable values of N or in terms of memory. It is also unable to use the memory bandwidth that will be available in this time frame. The growth in memory for the three architectures is summarized in Table 2. Narrow values of P for the parallel-pipelined version have significant disadvantages as the number of units increases. At equivalent bandwidths, the parallel-pipelined approach does not outperform the parallel approach until N is over 2048.

For a "slightly more than doubled" FPGA, there are two options for the parallel architecture. It can have 16 parallel units with full decoupling with a BW of 8 (4 lead-in stages, 8 units wide). Alternatively, it can have 32 parallel units with no lead-in stages. Similarly, the pipelined and parallel-pipelined implementations can have 32 butterfly units each. The case of the 32 parallel units gives significantly better performance than the 16 fully decoupled units because all of the units are used in each iteration rather than having the lead-in units sit idle for much of the computation. What is unique at this FPGA size is that the parallel and parallel-pipelined implementations have the same number of units. In general, the parallel approach is limited in that it is constrained to a parallelism that is a power of 2 (or, a power of the radix). Thus, only in rare circumstances will the parallel approach be able to use as many units as the parallel-pipelined approach. When it does, however, Figures 8(e) indicates that the performance of the parallel approach is competitive with, and sometimes better than, the parallel-pipelined approach.

7.3. FPGAs vs. CPUs

Figure 9 compares estimates of performance for FPGAs and CPUs over the next few years. The data for the 2.8 GHz Pentium-4 is from [7] using the Intel MKL. The 3.8 GHz Pentium-4 data was estimated using the ratio of the clock frequencies. Beyond that, CPUs are assumed to double in performance every 18 months, following the widely accepted corollary to Moore's Law (starting with the 3.8 GHz Pentium-4). Initial versions of the prototypes on FPGAs indicate that the design will run at 160 MHz on a Virtex-2Pro 100-6 and FPGAs are assumed to double in clock frequency and area every two years following historical trends[20].

The size of a single FFT must currently approach 8K items for the FPGA to outperform a microprocessor and microprocessors currently have a dramatic advantage for small FFTs. Within two years, the FPGAs should have an advantage for FFTs as small as 1024 items, and within 4 years, that advantage is expected to be dramatic for larger sizes.

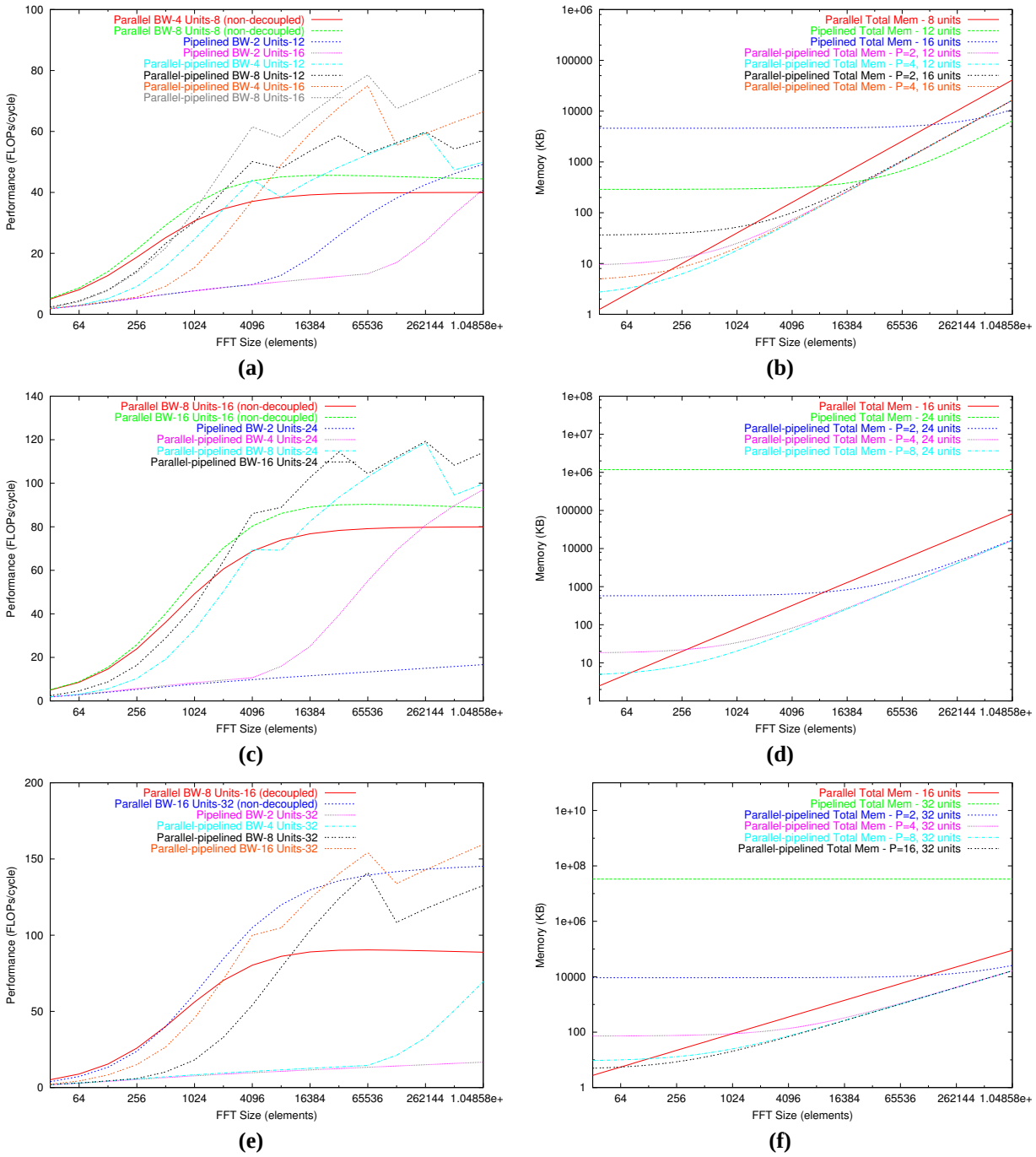


Figure 8. A comparison of the **(a,c,e)** performance and **(b,d,f)**memory requirement of FFT implementations

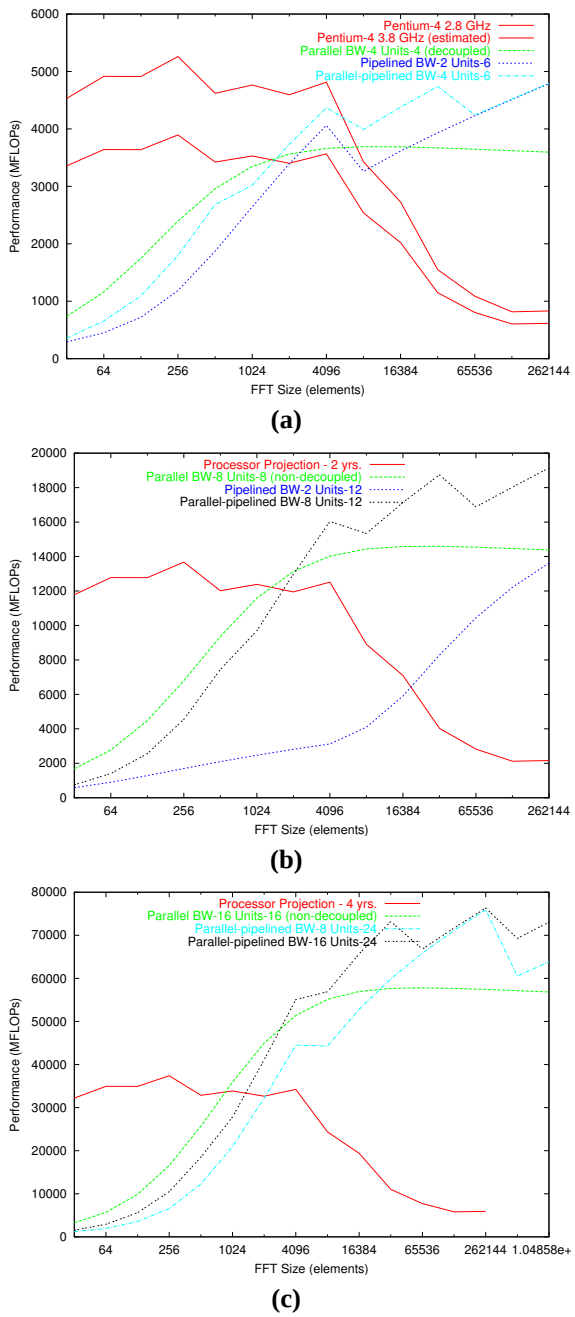


Figure 9. FPGAs vs. CPUs on current technology **(a)**, in two years **(b)**, and in four years **(c)**

Note that the far right hand side of the graph is not particularly fair since FPGAs do not have the internal memory to achieve that level of performance. Also, the elbow in the CPU curve will move to the right in future generations as the cache size grows; however, the comparison does not change as the CPU curve is still flat at 1024 elements while the FPGA performance is still growing.

7.4. Prototyping

Prototypes of the parallel and pipelined FFTs were developed to validate the analysis here. These prototypes were tested on the Annapolis Microsystems Wildstar-II Pro board using a single Virtex-2Pro 100 and 4 banks of DDR-SRAM. All memories were accessed in such a way as to mimic SDRAM. Four complex, double-precision floating-point butterfly units will fit in the Virtex-2Pro along with a two-by-two lead-in unit to provide full decoupling for the parallel array. The pipelined version of the FFT can place 6 full butterfly units in the Virtex-2Pro 100.

8. Conclusions

The FFT exposes some limitations of double precision floating-point arithmetic on FPGAs, but it still achieves good performance the larger transforms needed to leverage an attached co-processor. The "right" way to achieve that performance depends on the size of the FFT, the size of the FPGA, and the memory bandwidth available. Increasing either the memory bandwidth or the size of the FPGA generally favors designs with wider, rather than deeper, pipelines; however, width and depth must be balanced to maximize the number of floating-point units on a chip.

Relative to processors, FPGAs are heavily penalized by their low clock rate and high latency floating-point units. These factors make current FPGAs inferior to microprocessors for a single, small FFT. For large FFTs, FPGAs show the same trend toward dramatically outperforming microprocessors that has been seen in other work[20, 21]. The pipelined architecture could currently address the limitations with short FFTs by streaming several such FFTs through the FPGA. Going forward, the pipelined-parallel architecture could also help; however, the total amount of FFT computation that is required to keep the large number of floating-point units busy will continue to grow.

9. Future Work

The double precision floating-point FFT on FPGAs has performance limitations for short vectors due to the floating-point unit latency. Future work will investigate mitigating this effect by streaming multiple, independent FFTs to the

circuit — a common operation in multi-dimensional FFTs. Furthermore, we will be studying better ways to leverage the many ports provided by block RAMs rather than having to replicate twiddle factor storage to provide additional twiddle factor bandwidth to the butterfly units.

The next step in studying double precision floating-point performance on FPGAs is to look at the ability to switch between various control structures to implement different portions of the algorithm. For example, an LU solver has both a matrix decomposition and a solve step. Switching from one to the other is reminiscent of the functional unit sharing approaches, but has a fundamental difference: all units will switch from one part of the algorithm to another at larger granularity. To study this, we will investigate the viability of algorithms such as LU solves and iterative sparse matrix solves using double precision floating-point on FPGAs.

References

[1] FPGA cores aim to beat DSPs at floating point, Nov. 2001. From webpage at http://www.electronicstalk.com/news/nal/nal103.html.

[2] Altera. Floating-point FFT processor (IEEE 754 single precision) radix 2 core, Jan. 2005. From webpage at http://www.altera.com/literature/wp/wp_fft_radix2.pdf.

[3] S. Choi, R. Scrofano, V. K. Prasanna, and J.-W. Jang. Energy-efficient signal processing using FPGAs. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 225–234, Monterey, CA, Feb. 2003.

[4] H. A. Chow, H. Alnuweini, and S. Casselman. FPGA-based transformable computers for fast digital signal processing. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 197–203, Napa Valley, CA, April 1995.

[5] M. deLorimier and A. DeHon. Floating point sparse matrix-vector multiply for FPGAs. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, Monterrey, CA, February 2005.

[6] C. Dick. Computing the discrete fourier transform on FPGA based systolic arrays. In *Proceedings of the 1996 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 129–135, Monterey, CA, Feb. 1996.

[7] M. Frigo and S. G. Johnson. Fft benchmark results, Jan. 2005. From webpage at http://http://www.fftw.org/speed/p4-2.8GHz-new/.

[8] G. Govindu, S. Choi, V. K. Prasanna, V. Daga, S. Gangadharpalli, and V. Sridhar. A high-performance and energy-efficient architecture for floating-point based lu decomposition on fpgas. In *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW)*, Santa Fe, NM, April 2004.

[9] P. Graham and B. Nelson. FPGA-based sonar processing. In *Proceedings of the 1998 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 201–208, Monterey, CA, Feb. 1998.

[10] A. H. Kamalizad, C. Pan, and N. Bagherzadeh. Fast parallel FFT on a reconfigurable computation platform. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03)*, pages 254–259, Sao Paulo, SP - Brazil, November 2003.

[11] S. J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics*, 117:1–19, 1995.

[12] S. J. Plimpton. Lammps web page, July 2003. http://www.cs.sandia.gov/ sjplimp/lammps.html.

[13] S. J. Plimpton, R. Pollock, and M. Stevens. Particle-mesh ewald and rRESPA for parallel molecular dynamics. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, Mar. 1997.

[14] QinetiQ. Quixilica high throughput fast fourier transform core, Jan. 2005. From webpage at http://www.transtech-dsp.com/datasheets/qx-fft_hm_v1.pdf.

[15] M. Shaditalab, G. Bois, and M. Sawan. Self sorting radix-2 FFT on FPGA using parallel pipelined distributed arithmetic blocks. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 337–338, Napa Valley, CA, April 1998.

[16] S. Shingu, H. Takahara, H. Fuchigami, M. Yamada, Y. Tsuda, W. Ohfuchi, Y. Sasaki, K. Kobayashi, T. Hagiwara, S. Habata, M. Yokokawa, H. Itoh, and K. Otsuka. A 26.58 tflops global atmospheric simulation with the spectral transform method on the earth simulator. In *Proceedings of the ACM / IEEE Supercomputing SC'2002 conference*, 2002.

[17] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155–162, 1995.

[18] W. W. Smith and J. M. Smith. *Handbook of Real-Time Fast Fourier Transforms*. Wiley-IEEE Press, New York, 1995.

[19] S. Sukhsawas and K. Benkrid. A high-level implementation of a high performance pipeline FFT on Virtex-E FPGAs. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI Systems Design (ISVLSI 04)*, pages 229–232, Lafayette, LA, February 2004.

[20] K. D. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, Monterrey, CA, February 2004.

[21] K. D. Underwood and K. S. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2004.

[22] L. Zhuo and V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on fpgas. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, NM, April 2004.

[23] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, Monterrey, CA, February 2005.