SYSTEMATIC GENERATION OF FPGA-BASED FFT IMPLEMENTATIONS

Hojin Kee[1], Newton Petersen[2], Jacob Kornerup[2], Shuvra S. Bhattacharyya[1]

[1]Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies,
University of Maryland, College Park, 20742, USA.
[2]National Instruments Corporation, Austin, 78759, USA.
{hjkee, ssb}@umd.edu, {newton.petersen, jacob.kornerup}@ni.com

ABSTRACT

In this paper, we propose a systemic approach for synthesizing field-programmable gate array (FPGA) implementations of fast Fourier transform (FFT) computations. Our approach considers both cost (in terms of FPGA resource requirements), and performance (in terms of throughput), and optimizes for both of these dimensions based on user-specified requirements. Our approach involves two orthogonal techniques — FFT inner loop unrolling and outer loop unrolling — to perform design space exploration in terms of cost and performance. By appropriately combining these two forms unrolling, we can achieve cost-optimized FFT implementations in terms of FPGA slices or block RAMs in FPGA, subject to the required throughput. We compared the results of our synthesis approach with a recently-introduced commercial FPGA intellectual property (IP) core — the FFT IP module in the Xilinx LogiCore Library, which provides different FFT implementations that are optimized for a limited set of performance levels. Our results demonstrate efficiency levels that are in some cases better than these commercial IP blocks. At the same time, our approach provides the advantages of being able to optimize implementations based on arbitrary, user-specified performance levels, and of being based on general formulations of FFT loop unrolling trade-offs, which can be retargeted to different kinds of FPGA devices.

Index Terms — Fast Fourier transform, Field-programmable gate arrays, Memory management, High-level synthesis.

1. INTRODUCTION

The fast Fourier transform (FFT) is one of the most widely-used and important signal processing functions, for example, in applications related to digital communications and image processing. Since the computational complexity of the FFT is $O(N\log N)$, where N the number of inputs, the FFT potentially requires multi-cycle processing, and can become a major bottleneck for overall system performance. To relieve this bottleneck, many commercial IP blocks provide a *streaming* form of the FFT with single-cycle-per-sample throughput. This high-throughput form of FFT comes at the expense of increased hardware cost, which in turn can lead to costly, over-designed hardware in situations where single-cycle-per-sample throughput is not required — that is, in situations where the FFT bottleneck is significant, but not so severe as to require such a high degree of throughput optimization.

This paper develops a systematic approach for generating a cost-efficient, FPGA-based FFT implementation based on a designer-specified throughput requirement. Our approach carefully integrates two orthogonal methods for trading-off hardware cost and performance. The first method, which can be viewed as outer loop unrolling of the targeted FFT, realizes parallelism by instantiating multiple processing cores (dedicated hardware subsystems) across FFT butterfly stages. The second method, which can be viewed as unrolling of the FFT inner loop, allocates multiple cores within each stage. Each of these methods has advantages and drawback compared to the other, and in general, an integrated application of both methods can lead to a more cost-effective solution for a given throughput constraint — e.g., a more cost-effective solution compared to a solution that applies only one of these methods, or that is based on a the high performance / high cost streaming FFT implementation. Furthermore, depending on the given throughput constraint, one of these unrolling methods may be of more critical utility than the other.

Motivated by these observations, we develop a comprehensive approach to mixing and matching outer and inner-loop unrolling for cost-efficient, throughput-constrained synthesis of FPGA hardware. In FPGA synthesis, slices (basic logic cells) and block RAMs (BRAMs) are limited, and usage in terms of these two resources is important in evaluating hardware cost [3]. Our synthesis approach is prototyped in National Instruments LabVIEW FPGA 8.5. LabVIEW is a graphical, dataflow-based programming environment for embedded systems design. LabVIEW features for HDL (hardware description language) synthesis and fixed point data types, along with LabVIEW's dataflow orientation make the tool well-suited to FPGA-based design of signal processing applications. The output of our techniques for synthesis and optimization of FFT configurations is a LabVIEW dataflow diagram that specifies the structure and functionality of an optimized FFT configuration. This diagram is then synthesized to an FPGA device by first invoking LabVIEW's HDL synthesis tool, and then mapping the resulting HDL code using the platform-specific tools of the targeted FPGA. In our experiments, we have targeted the Xilinx Virtex II Pro FPGA.

In our experiments, we have compared the targeted cost metric — the usage of FPGA slices and BRAMs — between implementations generated by our novel synthesis flow, and those obtained from the Xilinx LogiCore library for identical levels of throughput. The results demonstrate that our synthesis approach provides results that are of similar cost to those from the commercial LogiCore library. This is encouraging since our approach provides the unique advantage of being synthesis-driven (as opposed to library-based) so that it can be driven by arbitrary performance levels rather than being restricted to a pre-determined subset of FFT configurations. Also, because it is based on an abstract synthesis formulation, it can be retargeted to different FPGA devices — e.g., by weighting or otherwise revising the cost function in terms of the resources that are most critical for a particular target.

In section 2, we briefly describe background related to the FFT algorithm, and related work on VLSI implementations for the FFT. Section 3 presents details of the two types of unrolling techniques that are applied in our approach to achieve throughput improvement. Section 4 analyzes each unrolling technique in terms of hardware costs, and discusses the issue of strategically integrating

both unrolling techniques to optimize the cost. Section 5 illustrates the result of generated FFT and comparison with Xilinx IP. Section 6 provides a conclusion of this paper.

2. BACKGROUND AND RELATED WORK

The discrete Fourier transform (DFT) for N points is given by

$$X_k = \sum_{i=0}^{N-1} x_i \cdot W_N^{ik}, \quad (1)$$

where

$$W_N^{ik} = \exp(-2\pi ik/N), \text{ and } k = 0, 1, \ldots, N-1. \quad (2)$$

The computational complexity of the DFT is $O(N^2)$. The radix-2 decimation-in-time fast Fourier transform (FFT) algorithm, illustrated in Figure 1, is widely used to compute the DFT with a complexity of $O(N\log N)$ [1]. Note that all logarithms n this paper have an implicit base of 2. To implement the FFT algorithm in a hardware, it is required to run a butterfly operation iteratively. In implementing the FFT algorithm, careful memory management is one important issue.

Ma [2] developed an efficient method for in-place memory management in FFT implementation. In Ma's approach an in-place strategy is employed to store butterfly outputs in the same memory locations that are used by the inputs to the butterfly. Such an in-place strategy is useful in reducing memory requirements, and enabling pipelining in terms of memory reads, butterfly operations, and memory writes. However, Ma's scheme is developed for an FFT core that involves a single butterfly unit, so the overall approach is limited in terms of throughput improvement. Nordin et al. [4] presented a parameterized soft core generator for the FFT based on the Peace FFT algorithm with the stride permutation approach proposed by Takala et al. [5]. By running multiple butterflies simultaneously with a scalable stride permutation, the generated FFT achieves an effective balance between hardware costs and performance features, and is also customizable based on given design constraints. Jackson et al. [6] proposed a systolic structure to provide for high throughput FFT implementation.

A distinguishing aspect of the approach that we develop in this paper is the realization of data parallelism with a carefully-configured address generator, and the integration of this address generation approach with an inner loop unrolling technique. This is in contrast, for example, to introducing special permutation structures for butterfly operations. Our approach, which is especially targeted to FPGA implementation, results in efficient utilization of FPGA slices.

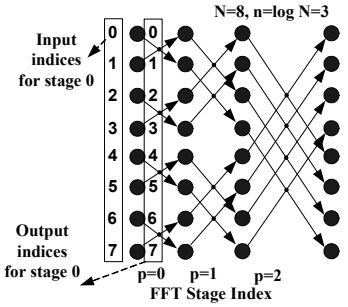

Figure 1. Signal flow graph of 8-point FFT with notational conventions illustrated. For each stage $p < (n-1)$, the data written through an output index for stage p corresponds to the data read through an input index for stage $(p+1)$.

3. UNROLLING TECHNIQUES

The radix-2 FFT algorithm involves running the butterfly operation iteratively. Using an in-place memory management scheme, we roll the butterfly operations within a given stage using a *for*-loop, which we refer to as the *inner loop*. Across different stages, we then employ another *for*-loop, which we call the *outer loop*. A *basic FFT core* (BFC) provides dedicated hardware for one butterfly operation, and we can execute a BFC iteratively with the aforementioned inner and outer *for*-loops to achieve a complete FFT implementation. However, rather than instantiating just one BFC for computing all FFT stages, we can achieve k times throughput improvement by running k BFCs simultaneously across stages, or by incorporating parallelism inside the BFC so that multiple butterfly operations can be executed in parallel within a given stage. We propose two orthogonal unrolling techniques to allocate and utilize BFCs in an efficient and scalable manner on FPGAs. The techniques have different cost functions in terms of usage of FPGA slices or BRAMs, and we show that in general, the two approaches should be considered jointly for cost-efficient FPGA-based, FFT implementation.

3.1 Outer Loop Unrolling

The iteration count for the outer for-loop in the FFT is equal to the total number of stages, $\log N$. Unrolling the outer loop by an *unrolling factor* $k > 1$ instantiates k "sub-FFT" cores. $(k-1)$ of of these sub-FFT cores have $\lceil \log N/k \rceil$ loop iterations each, while the remaining one has $(\log N - \lceil \log N/k \rceil (k-1))$ iterations. The design of a sub-FFT core is identical to that of the BFC design described in [2], except for some initialization details, and the iteration count. In this approach, k sub-FFT cores are running in parallel, and up to a factor of k improvement in throughput can be achieved. This approach introduces k identical copies of the sub-FFT core, so that it is expected that a factor of k increase in hardware cost results — in terms of BRAMs and FPGA slices. The trade-offs associated with outer loop unrolling are complemented by inner loop unrolling, which we elaborate on in the following section.

3.2 Inner Loop Unrolling

While unrolling the outer loop is realized by adding more copies of the sub-FFT core, we unroll the inner loop by executing multiple butterfly units in parallel inside a sub-FFT core. That is, we parameterize the sub-FFT core with the number of hardware butterfly units, and we increase the value of the associated parameter to trade-off increased area for improved throughput. If in-place memory management is used, then k butterfly units within a sub-FFT core require $2k$ independent (parallel) data memory banks (DM banks); however, the amount of storage required in each DM bank is reduced by a factor of k so that the total amount of DM bank storage required after inner loop unrolling is unchanged compared with a sub-FFT core that has a single butterfly unit. Note that in an FPGA device, each DM bank will normally be implemented by one or more BRAMs [3].

If only a single read-port is available in each DM bank, then to simultaneously read two corresponding input values X_u and X_l before a butterfly operation, the two locations must be stored in

different DM banks. Ma [2] determined that the indices of two inputs, u and l, for a butterfly unit in the p th stage are identical, except for the p th bit in their binary patterns. Thus, if we have two DM banks for the butterfly inputs, and if the p th bit of the input index is used to select the bank, then the inputs will always arrive from different memory banks, which means that they can be read in parallel. Moreover, if we derive the memory address by simply discarding the p th bit of each input index, and taking the remaining bit pattern, then it is possible to access both inputs for each operation from the same address in the two DM banks. This reduces the logic needed for address generation.

If x denotes a binary bit pattern, and y denotes a non-negative integer, let $RL(x, y)$ denote the bit pattern that results from left-rotation of x by y bit positions, and similarly, let $RR(x, y)$ denote the bit pattern that results from right-rotation of x by y bit positions. Also, for bit patterns x_1 and x_2, let $CONCAT(x_1, x_2)$ denote the concatenation of x_1 and x_2. For example, if $x_1 = 110$, and $x_2 = 01100$, then $RL(x_1, 2) = 011$, $RR(x_2, 3) = 10001$, and $CONCAT(x_1, x_2) = 11001100$.

For efficiency in hardware utilization, we restrict the inner loop unrolling factor to be a power of 2; that is, $k = 2^r$ for some non-negative integer r. Given an inner loop unrolling factor $k = 2^r$, there are k hardware butterfly units in each parameterized sub-FFT core, and $2k$ DM banks (two for each butterfly unit). Let these DM banks have indices $0, 1, \ldots, (2k-1)$. Each DM bank contains $2^{(n-r-1)}$ data locations that are accessed during FFT operation, where $n = \log N$, and N is the number of sample points involved in the overall FFT computation. Suppose that p is the index of a given FFT stage (i.e., $0 \le p \le n-1$); let $B^p = b_r b_{r-1} \ldots b_0$ be the binary bit pattern of some DM bank index in this stage; and let $A^p = a_{n-r-2} a_{n-r-1} \ldots a_0$ be the bit pattern for some DM bank address that is accessed in this stage. For clarity, our conventions for input indices, and FFT stage indices, as well as N and n are illustrated in Figure 1. From the above definitions, the input index that corresponds to address A^p in our memory management scheme can be derived as

$$u = RL(CONCAT(RR(A^p, p), B^p), p) \quad (3)$$

$$= a_{n-r-2} a_{n-r-3} \ldots a_p b_r b_{r-1} \ldots b_0 a_{p-1} a_{p-2} \ldots a_0 . \quad (4)$$

With this notation, the least significant bit (LSB) in a given DM bank index b_0, represents the p th bit of the corresponding input index in the p th FFT stage. Since two input indices for a given butterfly operation in the p th stage are the same except for the p th bit, the input index u and the index l for the other input in the same butterfly operation have their data stored in DM banks $b_r b_{r-1} \ldots b_1 0$ and $b_r b_{r-1} \ldots b_1 1$, respectively. These two DM banks, whose indices are identical except for their LSBs, are a pair of DM banks that are assigned to the same hardware butterfly unit. Thus, we entirely avoid any selection logic between DM banks and butterfly units. Moreover, two inputs to a given butterfly operation can be read from the same address because the corresponding input indices are identical, except for the p th bit, and the p th bit is the one that the selects the DM bank for a given butterfly unit.

After a butterfly operation in the p th stage, the output should be written to a DM bank so that it will be ready for the read in the $(p+1)$ th stage. In other words, the destined DM bank index and the address for writing back an output indexed by u in the p th stage are equivalent, respectively, to the DM bank index and the address for reading the input indexed by u in the next stage, stage $(p+1)$. Thus, the destined DM bank index and its associated address for writing butterfly output data can be generated by an inverse mapping from (4) with output index u and stage index $(p+1)$. This inverse mapping is given by

$$B^{p+1} = a_p b_r b_{r-1} \ldots b_1 , \text{ and} \quad (5)$$

$$A^{p+1} = a_{n-r-2} a_{n-r-3} \ldots a_{p+1} b_0 a_{p-1} a_{p-2} \ldots a_0 . \quad (6)$$

The address, A^p, can be generated efficiently by

$$A^p = RL(Counter, p) . \quad (7)$$

Here, the value of $Counter$ is increased by one every clock cycle, so that bit a_p in A^p is flipped on each clock cycle. This provides a resource-efficient mechanism for generating a_p, and (via (5)), generating the required sequence of B^{p+1} selections.

4. COST/PERFORMANCE ANALYSIS

The two orthogonal unrolling techniques developed in the previous section exhibit different profiles of FPGA resource consumption. While outer loop unrolling pipelines multiple FFT cores, inner loop unrolling executes multiple butterfly units in parallel inside a single FFT core. Since the inner loop unrolling technique involves more localized control (i.e., control over a single FFT core) it generally consumes less FPGA logic resources compared with the more extensive control structures needed for outer loop unrolling. However, inner loop unrolling is less flexible in terms of the set of possible unrolling factors — to preserve the applicability of our streamlined approach for inner loop memory management, the inner loop unrolling factor must be a power of two. This requirement makes the range of achievable speedups for the inner loop unrolling technique to be limited to powers of two, while outer loop unrolling can be applied with arbitrary positive integer factors. Thus, for example, if the degree of speedup required to achieve the given throughput constraint is not a power of two, then a combination of inner-loop and outer-loop unrolling may lead to the most cost-effective solution.

Figure 2 shows FPGA slice and BRAM utilization as functions of the unrolling factor for both inner and outer loop unrolling. These results are obtained after synthesis, and include the streamlining effects of our proposed schemes for address generation and memory management. For both kinds of unrolling, BRAM and FPGA slice utilization increase linearly with the degree of speedup achieved (unrolling factor). Also from Figure 2, we see that inner loop unrolling is more area-efficient compared to outer loop unrolling for the same throughput increase. However, recall that inner loop unrolling is restricted to factors that are powers of 2. In increasing FFT length, we take advantage of more fully using BRAMs in a wider range of inner loop unrolling factors.

For use in analytical design space exploration, the following cost functions can be derived from these synthesis results:

$$u_{inner} = s_{inner} \cdot u_{initial}(k_{inner} - 1) + u_{initial} , \text{ and} \quad (8)$$

$$u_{outer} = s_{outer} \cdot u_{initial}(k_{outer} - 1) + u_{initial} . \quad (9)$$

Here, u_{inner} and u_{outer} are the amounts of utilization (FPGA slice or BRAM utilization) after inner and outer loop unrolling, respectively; $u_{initial}$ represents the amount of resource utilization without any unrolling; k_{inner} and k_{outer} are inner and outer loop unrolling factors, respectively; and s_{inner} (s_{outer}) is a constant

factor that represents the slope of the linear plots for inner (outer) loop configurations in Figure 2.

The cost functions in (8) and (9) are for inner and outer loop unrolling in isolation. If both forms of unrolling are applied in combination, then the total hardware resource requirements can be expressed as

$$u_{combined} = s_{outer} \cdot u_{inner}(k_{outer} - 1) + u_{inner}, \quad (10)$$

where u_{inner} is derived as in (8). The speedup resulting from such a combination can be expressed as

$$k_{combined} = k_{inner} \cdot k_{outer}. \quad (11)$$

Given a throughput constraint, (10) and (11) can be used to efficiently search the space of feasible designs (i.e., designs with satisfactory throughput) for a cost-optimal solution. In particular, candidate pairs (k_{inner}, k_{outer}) that satisfy the throughput constraint (based on (11)) can be evaluated to select the one that minimizes cost (based on (10). This evaluation can be pruned by noting that whenever a particular pair (k_{inner}', k_{outer}') is found to satisfy the throughput constraint, we need not consider any additional pairs $(k_{inner}'', k_{outer}'')$ such that $k_{inner}'' \geq k_{inner}'$ and $k_{outer}'' \geq k_{outer}'$ are both satisfied. This approach allows for very rapid, pre-synthesis determination of cost-effective architectures for given throughput constraints.

5. EXPERIMENTAL RESULTS

We have targeted the Xilinx Virtex II Pro P30 embedded in the National Instruments PCI-5640R to synthesize implementations derived by our architecture generation techniques for the FFT. Figure 3 shows additional synthesis results from FFT implementations derived by our proposed techniques. The specific form of FFT implemented in these results is a radix-2 FFT with 2048 samples, with each sample represented as a 16-bit, fixed-point value.

Figure 3 reports the FPGA resource utilization when the target speedup is 6. Note that $4k$ 18x18 multipliers are used under an unrolling factor of k. We use a target speedup of 6 here because the throughput of a sequential implementation (no unrolling) on this device is 5.5 cycles per sample, and 6 is the lowest integer speedup needed to achieve the common "streaming FFT" target of 1 cycle per sample. Using the high level exploration approach developed in Section 4, and the device-specific slopes and initial

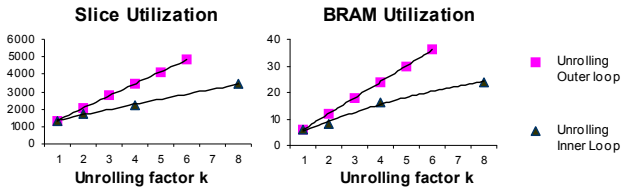

Figure 2. Synthesis results from loop unrolling.

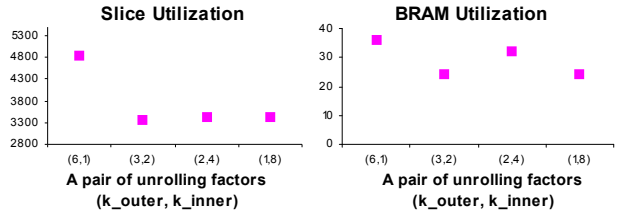

Figure 3. Synthesis report from the combined unrolling technique when the target speedup is 6. (N=2048)

utilizations from the curves in Figure 2, we can calculate analytically that when (k_{outer}, k_{inner}) is equal to $(3, 2)$ and $(1, 8)$, respectively, then the generated FFT core is optimized in terms of FPGA slice usage and BRAM utilization. These results agree with the optimal values observed from the two curves from actual synthesis results in Figure 3, thereby demonstrating the accuracy of our high level exploration method. To compare our approach with relevant commercially-available FFT core, we evaluated the FFT core that is available from the Xilinx LogiCore library under the two different throughput levels that are available for it — streaming throughput and sequential (resource-optimized) throughput. For streaming FFT performance (one cycle per sample throughput), our approach required 23% less FPGA slices compared to the Xilinx core, but 140% more BRAMs. For the sequential performance level, our approach required 30% fewer slices, and 17% more BRAMs. Note that the latter comparison ("sequential performance") does not include any unrolling, and is therefore essentially a comparison with Ma's FFT configuration, which is the special case of our approach that results when no unrolling is carried out.

6. CONCLUSION

In this paper, we have developed a systematic approach for generating dedicated FFT subsystems for FPGA implementation. Our approach incorporates efficient FFT address generation and memory management, and applies two orthogonal loop unrolling methods to provide a tunable trade-off between performance and FPGA resource costs. We also develop an analytical approach for high level design space exploration, which allows one to estimate the most resource-efficient FFT architecture configuration for a given throughput constraint and a given critical target resource (e.g., FPGA BRAM or logic slices). Our methods are demonstrated through extensive synthesis experiments using the Xilinx Virtex II Pro FPGA device family. Our synthesis results quantify the cost-performance trade-offs in our proposed class of FFT architectures. A distinguishing characteristic of our approach, compared to commercially available FFT IP cores and other specialized FFT implementations, is that we provide a systematic method to generate an FPGA-based FFT architecture while taking into account trade-offs between performance and cost.

7. REFERENCES

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Mathematics of Computation, Vol. 19, No. 90, 297-301, 1965.

[2] Y. Ma, "An Effective Memory Addressing Scheme for FFT Processors," IEEE Transactions on Signal Processing, vol. 47, Issue 3, pp. 907-911, March 1999.

[3] W. Wolf. *FPGA-Based System Design*. Prentice Hall, 2004.

[4] G. Nordin, P. A. Milder, J. C. Hoe, M. Puschel, "Automatic Generation of Customized Discrete Fourier Transform IPs", Design Automation Conference, pp. 471- 474, 2005.

[5] J. Takala, T. Jarvinen, P. Salmela, and D. Akopian. Multi-port interconnection networks for radix-r algorithms. In Proc. IEEE Intl. Conf. Acoustics, Speech, Signal Processing, 2001.

[6] P. A. Jackson, C. P. Chan, J. E. Scalera, C. M. Rader, and M. M. Vai, "A Systolic FFT Architecture for Real Time FPGA Systems", High Performance Embedded Computing Workshop, 2004.