High-Throughput Implementation of a Million-Point Sparse Fourier Transform

Abhinav Agarwal, Haitham Hassanieh, Omid Abari, Ezz Hamed, Dina Katabi & Arvind
Computer Science & Artificial Intelligence Laboratory, Massachusetts Institute of Technology, MA, USA
Email: {abhiag, haitham, abari, ezz, dina, arvind}@csail.mit.edu

***Abstract*—The emergence of data-intensive problems in areas like computational biology, astronomy, medical imaging, *etc.* has emphasized the need for fast and efficient very large Fourier Transforms. Recent work has shown that we can compute million-point transforms efficiently provided the data is sparse in the frequency domain. Processing input samples at rates approaching 1 GHz would allow real-time processing in several such applications. In this paper, we present a high-throughput FPGA implementation that performs a million-point sparse Fourier Transform on frequency-sparse input data, generating the largest 500 frequency component locations and values every 1.16 milliseconds. This design can process streamed input data at 0.86 Giga samples per second, and does not make any assumptions of the distribution of the frequency components beyond sparsity.**

I. Introduction

Processing million-point Fourier Transforms in real time can enable numerous applications ranging from GHz-wide spectrum sensing and radar signal processing to high resolution computational photography and medical imaging. Increasing the size of the transform is important for precisely pinpointing/localizing the sparse signals present in large frequency bands. Using a fast million-point transform allows such localization in a single step at a fast refresh rate. As an example, such a design could be used for real-time tracking of a narrow-band rogue transmission that is rapidly frequency-hopping across a wide-band range. Today, efficient million-point Fast Fourier Transforms (FFTs) are not practical. Hardware implementations of such large FFTs are prohibitively expensive in terms of high energy consumption and large area requirements. However, for most of the above applications the Fourier transform is sparse which means that only few of the output frequencies have energy and the rest are noise. Recent work [1] in the field of algorithms has shown how to compute these sparse FFTs (SFFT) in sub-linear time more efficiently than standard FFTs using much smaller number of samples than what is needed to compute the full FFT.

In fact, there are several SFFT algorithms which make different assumptions about the input [2]. A hardware implementation of a different version of SFFT was recently published [3]. Our FPGA implementation differs from [3] in two key aspects. First, the design in [3] is customized for a specified signal size whereas our implementation provides configurable parameters and hence can be adopted by various applications. Second, the algorithm underlying our implementation is more robust to noise. Specifically, [3] identifies active frequencies by computing exact phase rotation which is very sensitive to noise. In contrast, our approach identifies active frequencies by comparing values and hence it can tolerate a significant amount of noise. Further, it does not require the user to provide a noise threshold. This algorithm is consequently more computationally and memory intensive with additional steps of max-selection, voting and book-keeping of extremely large data vectors. This algorithm has been implemented in software, but the published implementations [4], [5] are unable to achieve high input data rates. These implementations are also not efficient from the perspectives of power, energy, unit cost or form factor.

For some of the applications, e.g., detection of frequency hopping transmissions, it is necessary to accept the input data in a streaming manner. This complicates implementation because the SFFT algorithm has been described using large shared data structures on which fast computation and comparisons need to be made. For hardware implementations, operations on such large data structures are impractical. This paper presents the first FPGA-based design of SFFT that works on streaming data at GHz rates. In particular, we can process million points of frequency-sparse streaming data to generate the locations and values of the largest 500 frequency coefficients in 1.16 milliseconds. This translates into input data rates of 0.86 Giga samples per second. The main contributions of our work include the following:

- The first dataflow description of the SFFT algorithm. The challenge here was to convert a control-centric description of the algorithm to a data-centric version.
- Determining the appropriate parameters of the algorithm to achieve our target performance goals without sacrificing accuracy and yet fit within the limited FPGA resources. These algorithmic parameters include the size of *dense* FFT to be used for a given sparsity level of the input and the number of iterations needed to identify all sparse frequencies.
- Novel hardware structures for Voter and Value Compute modules (see section III). For other modules such as Dense-FFT and Selector, we modified known hardware structures.
- A parameterized architecture so that the design can be used in various frequency-sparse applications.

In Section II, we give a brief overview of the SFFT algorithm that serves as the background for our implementation. In Section III we describe the design architecture of various component modules and how they interact with each other. In Section IV, we present the FPGA implementation results, discuss resource usage and performance of the design, and compare with existing software implementations. Finally, we present our conclusions in Section V.

II. Background

SFFT is an iterative algorithm where in each iteration, the million N frequencies are mapped to B buckets. This is done by performing a B-point FFT on a sample of B input points. The energy or the value of each bucket is the sum of the values of the N/B frequencies that are mapped into the bucket. By carefully choosing the input samples, one can ensure that in each iteration different frequencies will map into different buckets with high probability. The algorithm repeats the bucketization process with a permuted set of input time samples. This results in a permutation of the frequency components and randomizes the mapping of frequencies to buckets [1].

Given an N-dimensional vector x, let x_i denote the i^{th} element of the vector and let $\hat{x}$ denote its Discrete Fourier Transform (DFT). Then for some permutations y of x, it turns out that $\hat{y}$ is a permutation of $\hat{x}$. We call such permutations, inverse modulo permutations (IMP). We need to define invertible integers before defining IMP.

An integer σ is *invertible modulo* N, if there exists another integer σ^{-1} such that $(\sigma \times \sigma^{-1}) \pmod N = 1$. If N is a power of 2, then all odd integers are invertible modulo N. *Inverse modulo permutation*, P_σ, is defined with respect to an integer σ that is invertible modulo N. For a given vector x, let y denote $P_\sigma(x)$. Then $y_i = x_j$ with $j = (\sigma \times i) \pmod N$. For such a pair of x and y, permutation in the time domain corresponds to inverse modulo permutation $P_{\sigma^{-1}}$ in frequency domain, i.e., $\hat{y}_i = \hat{x}_j$ where $j = (\sigma^{-1} \times i) \pmod N$.

For each iteration, we select different random values for σ from the set of invertible integers, to get different mappings of frequencies into buckets. As an example, consider a 64-point input dataset whose frequency-transform has a single non-zero frequency component, whose frequency index and magnitude needs to be determined. When the sub-sampled input goes through an 8-point FFT, the non-zero component would land in one of 8 buckets, whose value would be higher than all other buckets. This bucket would have 8 candidate frequencies, one of which is the actual non-zero component. For the chosen example, we have two bucketization iterations with selected values of σ as 17 and 55. The values of sigma are randomly selected, as long as each is invertible modulo 64. Given these values, each frequency index gets mapped to one of 8 buckets, as shown in Table I.

Figure 1 illustrates how the non-zero frequency component is located by determining the intersection of candidate frequencies from each iteration. Bucket B_1 in iteration 1 and bucket B_5 in iteration 2 are the high buckets. The only frequency index that is common to these two buckets is 25. Thus, 25 is the required non-zero frequency component. The SFFT algorithm describes this step as a voting process, where all candidate frequencies landing in high buckets receive a *vote*. Non-zero frequency components receive votes in all iterations as they land in high buckets in every iteration. By tallying the votes after all iterations, frequency components with the most votes are the non-zero frequency components. The explanation we have provided in terms of set intersection is more suitable from an implementation point of view.

As the number of non-zero frequency components in the input increases, the number of high buckets in each iteration also increases and we need more iterations to determine the indices of the non-zero components. In the final step of the SFFT algorithm, the values of the located frequency components are estimated from the average of the values of the buckets that they are mapped into in each iteration.

Table I. Mapping input frequency indices into buckets for given sigma values for $N = 64$. Bucketization for further iterations is produced similarly.

Bucket Index	**Iteration 1** $(\sigma, \sigma^{-1}) = (17, 49)$	**Iteration 2** $(\sigma, \sigma^{-1}) = (55, 7)$
B_0	0,4,17,21,34,38,51,55	0,1,10,19,28,37,46,55
B_1	8,12,25,29,42,46,59,63	2,11,20,29,38,47,56,57
B_2	3,7,16,20,33,37,50,54	3,12,21,30,39,48,49,58
B_3	11,15,24,28,41,45,58,62	4,13,22,31,40,41,50,59
B_4	2,6,19,23,32,36,49,53	5,14,23,32,33,42,51,60
B_5	10,14,27,31,40,44,57,61	6,15,24,25,34,43,52,61
B_6	1,5,18,22,35,39,48,52	7,16,17,26,35,44,53,62
B_7	9,13,26,30,43,47,56,60	8,9,18,27,36,45,54,63

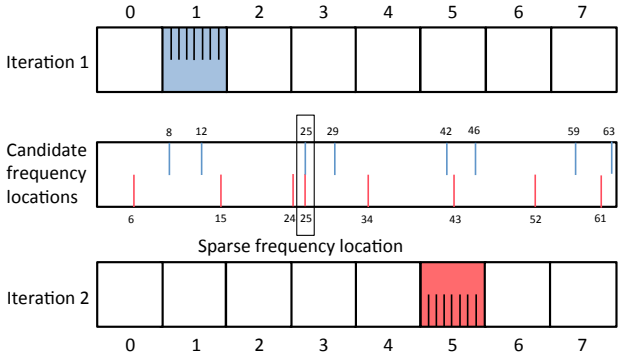

Fig. 1. The colored bucket indicates the bucket with high value. The chosen example has a single non-zero frequency component. Since B_1 of iteration 1 and B_5 of iteration 2 have only one frequency index in common, $\hat{x}_{25}$, it must be the sparse frequency component. If these buckets had more than one frequency in common, further iterations would be required for disambiguation.

III. Architecture

The SFFT implementation consists of several modules, each implementing a distinct stage of the algorithm. Figure 2 shows the various stages involved in the SFFT algorithm. The first two blocks implement the sampling and filtering stage, while the subsequent four blocks (highlighted in blue in Figure 2) are the SFFT Core. The sampling and filtering stage generates small data slices from the million-point input for the SFFT Core, as shown in Figure 3. The SFFT Core modules are responsible for the bulk of the computation and resource usage in the algorithm. In this paper, we will focus on the implementation of the Core modules with the first stage used as a pre-computation step. However, for better understanding of the algorithm, we first provide a brief description of the first stage and its implementation.

The sampling step involves minimal computation, and simply uses a pre-determined table of values[1] to select input samples from the million-point dataset to generate eight 8192-point input slices. Each of these slices is multiplied by a filter and then aliased in half to generate eight 4096-point input slices for the SFFT Core. The aliasing step increases robustness of the algorithm [1]. As long as the sampling step

[1]Selected samples are $x_{[\sigma k i] mod\ n}$, where $k = 1..8$ and $i = 1..8192$

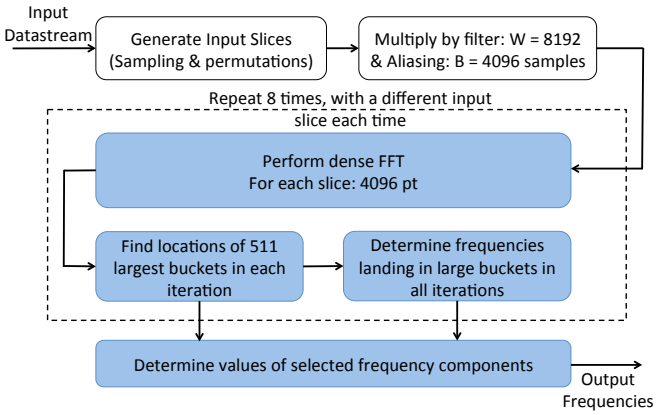

Fig. 2. Stages of the SFFT algorithm. Core stages are highlighted.

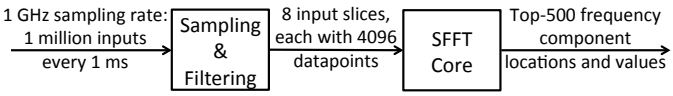

Fig. 3. Input-output characteristics for sampling and SFFT Core

Table II. Parameters for SFFT Core implementation

Parameter	Value
Input data type	Complex fixed-point
Fractional bits for fixed-point data	24
Total number of input data values	2^{20}
Maximum non-zero input frequency	500
Number of iterations in algorithm	8
Size of FFT in each iteration	4096

can supply input slices at the rate of computation by the SFFT Core, the overall throughput is unaffected. Figure 4 shows how this stage is implemented using stored address values for each input sample to be used[2]. The 3-bit *Repetitions* value indicates how many times a particular input sample is used, as it could be present in any number of the 8 input slices. The 13-bit position address is used to multiply the input sample with the corresponding filter time sample. The aliasing step is implemented by using only the 12 LSBs of the position address for storage and adding the previously stored value at the 12-bit address to filtered data sample.

Figure 5 shows the modules used in our implementation of the SFFT Core and their input-output semantics. Our design has been parameterized to allow design exploration and to generate optimized results for the desired specifications. The parameters chosen for the discussed implementation are given in Table II. We chose the input data type to be complex fixed-point with 24 fractional bits for each of the real and imaginary components. The high number of fractional bits ensures that we have sufficient accuracy for various applications. The number of input samples is 2^{20}, thus each input sample has a 20-bit location index and a 64-bit value (accounting for real and imaginary sign bit, integral bit and six overflow bits). The input data is constrained to have a maximum of 500 non-zero frequency coefficients. Increasing the number

[2]Maximum of 8×8192 input values are used. A separate 1-bit value for each of the million input samples indicates whether it is used.

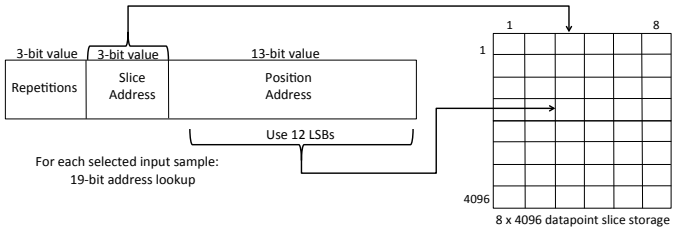

Fig. 4. Address lookup used for sampling and filtering input data points

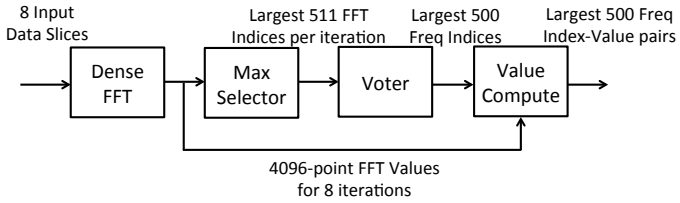

Fig. 5. Modules implementing the SFFT Core

of iterations and size of FFT in each iteration, increases the accuracy of the probabilistic SFFT algorithm. But it also increases the resource usage and time required for completion. We chose 8 iterations with a 4096-point FFT in each iteration, as this choice gave sufficient accuracy while still providing an achievable hardware target. Each module was targeted to achieve a minimum operating frequency of 100 MHz. We next describe the architecture of the SFFT Core modules in detail.

A. *4096-point FFT*

The SFFT algorithm requires taking a standard FFT of filtered input data slices, which we refer to as dense FFT. Our design of the dense FFT module was required to have a high throughput, low area and maximum size of the FFT possible. The larger the size of the FFT, the lower is the chance of collisions occurring due to non-zero frequency components being mapped to the same bucket. Initial attempts to use folded in-place FFT designs [6] failed, as they did not scale to a size beyond 512 points for the 24-bit input data. Instead, this implementation uses a fully pipelined streaming FFT architecture [7], utilizing a Radix-2^2 Single Delay Feedback. This design has a simple control structure with near-optimal storage and computation resource usage.

We extended the 256-point design cited in [7] to the required 4096-point FFT. To make this increase and allow efficient mapping to FPGA, we had to make several changes. We prevented the occurrence of a long critical path, by making each internal block of the FFT architecture a latency-insensitive parameterized module, designed as shown in Figure 6, with buffered input-output queues. *Latency insensitivity* implies that the depth of the queues does not affect correctness of the design. In the figure, N is a parameter that varies from 1 to 1024. The original design had a large single counter that controlled all stages synchronously. We split it into independent 2-bit counters for each stage. Figure 7 shows how the internal blocks are instantiated with appropriate parameter values to generate the pipelined 4096-point dense FFT implementation. We used pipelined multipliers for complex fixed-point arithmetic, and adequate buffering in FIFO queues between blocks to allow the design to continuously stream data

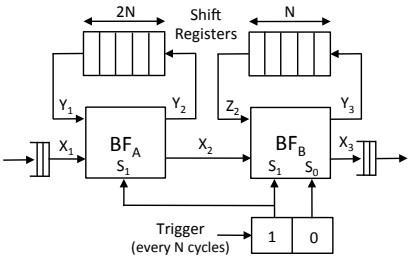

Fig. 6. Single parameterized block of streaming FFT

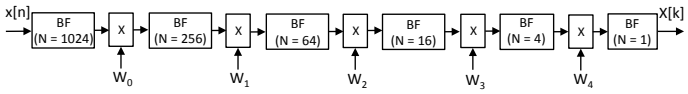

Fig. 7. Architecture of the 4096-point dense FFT

across iterations as an elastic pipeline. The twiddle factors W_i used in the computation were generated as a look-up table that each block can independently query for values. The indices for the twiddle factors are determined by the collective value of the 2-bit counters present in each block. Each block has two shift registers that map directly to FPGA shift registers. Absence of large multiplexers, which are usually present in folded in-place FFT designs, allows this implementation to be highly efficient in FPGA resource usage. The design is parameterized for the input data type, FFT size and amount of pipelining in the complex multipliers.

B. Max Selector: Buckets with maximum value

In the previous stage, by performing the 4096-point FFT we mapped the 2^{20} input frequencies into 4096 buckets. This stage determines which of these buckets have a large magnitude, indicating that one or more of the frequencies mapped to them are non-zero. Selecting buckets by setting a threshold would have been sensitive to noise levels in input and hence, not robust. Sorting all the FFT outputs to generate the ordered magnitudes was observed to be highly resource intensive and time consuming, as well as overkill since the algorithm does not require them to be ordered. Instead, we implement this step by selecting the largest (but unordered) 511 magnitudes of the 4096-point FFT output for each iteration. The chosen selector architecture operates on $2^n - 1$ entries, hence the number of entries being 511. Since the input data has a maximum of 500 non-zero frequency coefficients, selecting top 511 buckets by magnitude was sufficient.

Our design is based on a pipelined heap priority queue architecture [8]. This architecture is suitable for high-speed heap operations for structures with hundreds of entries. Figure 8 shows our implementation of the Top-511 element selector. We modified the ASIC architecture in [8] to allow for efficient mapping to FPGAs, including single port communication between stages of the heap, address tagging of entries, individual operation counters in each stage to cut critical path, use of *Register File* structures for storing values, and the method of generating the output as a bit vector. The magnitudes of the FFT outputs are tagged with their indices and inserted into a binary tree structure with 511 entries. Entries are addressed from 1 to 511, with the k^{th} entry being the parent of (and smaller in magnitude than) $2k^{th}$ and $2k + 1^{st}$ entries. Thus, the 1^{st} entry, called the root, has the smallest magnitude of

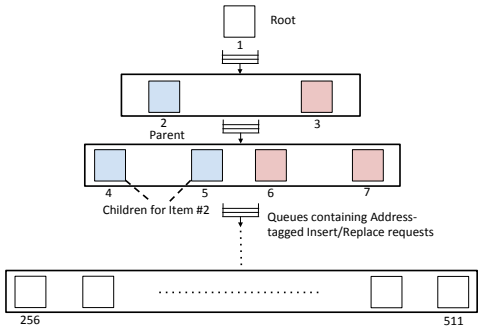

Fig. 8. Architecture of the Max Selector module

all the entries in the tree. The entries are divided into stages, each stage containing indices from 2^i to $2^{i+1} - 1$, where i goes from 0, for the root stage which has a single entry, to 8, for the last stage which has 256 entries. This allows write operations to be localized and prevents generation of large multiplexers during FPGA synthesis. In the first phase of operation, new elements are filled into the tree in ascending order of addresses, pushing larger entries down and maintaining the root as the minimum entry. The next address to be filled is appended as a tag with incoming entries.

Once the first 511 FFT outputs are inserted into the tree, the second phase starts with further streaming inputs being compared with the root to check whether they are larger than the root. Only if they are, they replace the root element and checks are done to maintain parent children relationships in the tree. For reducing the critical path, dependencies between adjoining stages were kept to a minimum and storage structures were made hierarchical. The queues between stages of the binary tree allow a pipelined design that can allow insertions into an unfilled tree every cycle, and replacements in a filled tree every alternate cycle. At the end of processing all 4096 FFT outputs, the tree contains the largest 511 magnitudes, each tagged with their corresponding location in the FFT output. The module output is a 4096-bit vector, with a high bit for each of the 511 selected FFT outputs.

C. Voter: Locating the top frequencies

This stage receives as input eight 4096-bit vectors with the large buckets indicated by the bits set to 1. Each of these buckets constitutes a set of 256 candidate frequencies that were mapped to this bucket using randomized permutations. This stage determines the frequency indices that have landed in top buckets in all iterations. The process can be understood as eight rounds of voting, with each iteration incrementing the votes of 511×256 distinct frequency indices and the final selection of indices with eight votes.

Implementing the stage as a naive iterative voting structure would have required book-keeping of nearly a million votes spread across a million candidate frequency locations. Implementing such a data structure in FPGA with the reading, writing and comparison of the votes for all candidates occurring within the required performance constraints is impractical. Since the values of σ chosen for each iteration are statically known, it is conceivable that a table of values can be generated that provides the static condition for each frequency index to be non-zero. But, this also runs into the issue of reading and comparing values from an extremely large data structure.

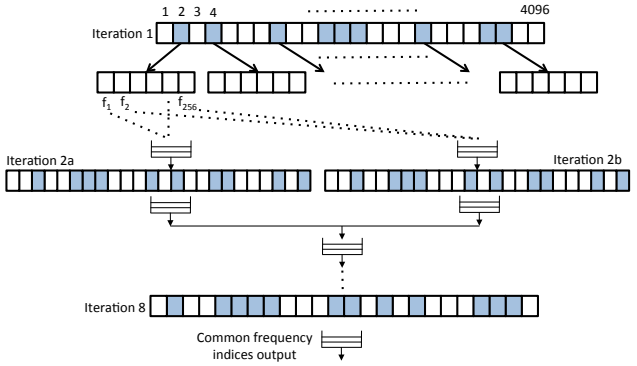

Fig. 9. Architecture of the Voter module implemented as a series of filters. Large buckets for each iteration have been highlighted in blue.

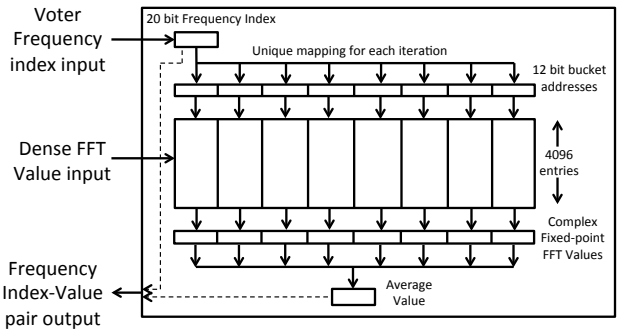

Fig. 10. Architecture of the Value Compute module

Instead, our implementation is a novel pipelined filtering process where we track candidate frequencies, instead of keeping track of votes. Candidate frequency indices are generated by the first iteration in a stepwise manner and are passed through seven filters. Each filter maps the incoming candidate frequency to the appropriate bucket for that iteration and checks if it lands in a top-511 bucket in the corresponding bit vector. If it does land in a large bucket, it is sent to the subsequent iteration otherwise it is discarded. The mapping functions are unique for each iteration and relate to the permutations used for generating the input data slices. At the end of the filtering process, only those frequency indices are passed through to the next stage which have been present in top buckets for all iterations. In order to further improve the throughput of this stage, we parallelized the processing of candidate frequencies by the second stage. For this, we duplicated the filter for the second iteration, seen as $2a$ and $2b$ in Figure 9, mapping 128 odd-indexed frequencies to $2a$ and 128 even-indexed frequencies to $2b$ for each high bucket in the first iteration. This decreased the number of cycles required to process the data by nearly 50% as most of the candidate frequencies get filtered out at the second iteration itself.

D. Value Compute: Magnitudes of top frequencies

This stage is responsible for computing the value of the top frequency components that have been determined by the previous stages. FFT outputs produced for all eight iterations are forwarded to the Value Compute module. They are stored locally in a Block-RAM memory structure with eight memory banks, which allows simultaneous processing of read requests from all eight banks.

The frequency locations determined to be non-zero in the input, obtained from the voter module, are mapped to their respective bucket locations for each iteration. Corresponding FFT outputs are read from the memory banks for each location. These eight bucket values for each selected frequency are then averaged to obtain the final value of the frequency components. Figure 10 shows the architecture designed for this module.

E. SFFT Core

The use of high-level latency insensitive interface specifications for component modules allows elegant plug and play generation of complex designs. This enabled us to quickly connect the various modules to generate the SFFT Core design. During design exploration, we had to modify the internal structure of various modules as a result of algorithmic modifications or for meeting resource and performance constraints. However, generating the overall design was straightforward once the component modules were complete. Adequate buffers were added to latency insensitive FIFO queues between modules to allow individual blocks to run at different rates without stalling the entire design. This was beneficial for the overall performance since several modules produced outputs sporadically and in a data-dependent manner.

IV. Implementation Results

The SFFT Core was designed using Bluespec SystemVerilog [9], a high-level hardware design language, that allows expression of parametrization and latency insensitive interfaces in the architecture. The design was synthesized, placed and routed for Xilinx ML605 platform with the target device as XC6VLX240T FPGA and the target clock frequency as 100 MHz. Table III shows the percentage Virtex-6 FPGA resource utilization of various modules of the implementation as well as the complete SFFT Core. Different modules have varying resource requirements due to the wide variety of designs. The Dense-FFT module has the highest DSP slice utilization, due to the use of pipelined multipliers for the complex fixed-point data. The Max Selector module has the maximum LUTs utilization, 18% of the total, due to the logic generated for reading and writing various entries in each internal stage. The Value Compute module has the maximum BRAM utilization of 14% for storing the FFT outputs for all eight iterations. Overall, as seen in the data, the Core fits well within the resources of a single Virtex-6 FPGA with 24% Slice Registers, 48% Slice LUTs, 26% BRAMs and 16% DSP48E utilization.

The design was mapped to the ML605 platform, and its performance was evaluated. The obtained results were compared with MATLAB results to verify correctness. Table IV shows the latency and throughput of various modules of the implementation as measured in FPGA clock cycles with an operating clock frequency of 100 MHz. The latency of the design is defined as the number of cycles taken from providing the first input data sample to a module to receiving the last output data sample from it. The throughput is defined, under steady state conditions with continuous input supply, as the number of cycles between the first output of the first input dataset to the first output of the second dataset. We have given the per-iteration and total number of cycles for the FFT and Max Selector modules. For the FFT module, the total latency is

Table III. FPGA Resource utilization, which is shown as a percentage of total resources available on Xilinx FPGA XC6VLX240T.

Design	Regs	LUTs	BRAMs	DSP48Es
Dense-FFT	3%	11%	8%	11%
Max Selector	8%	18%	0%	0%
Voter	10%	14%	3%	2%
Value Compute	1%	2%	14%	2%
SFFT Core	**24%**	**48%**	**26%**	**16%**

Table IV. Latency and throughput in FPGA clock cycles.

Design		Latency (cycles)	Throughput (cycles)
Dense-FFT	1 iteration	13682	6826
	Total	61468	54612
Max Selector (Avg)	1 iteration	5888	5888
	Total	47104	47104
Voter (Avg)	Total	68816	68816
Value Compute	Total	33788	33788
SFFT Core (Avg)	Total	**138646**	**116024**

less than $8\times$ the single iteration's latency because the pipelined architecture allows overlapping execution. The performance of the Max Selector and Voter modules is data-dependent. We have shown the average case numbers for these modules, and similarly for the entire SFFT Core design. For the SFFT Core, the total number of cycles per transform under steady state is significantly less than the sum of all individual components due to the pipelined nature of the architecture that allows overlapping computation between various modules.

We synthesized, placed and routed individual modules as well as the complete design to obtain processing times for each component. Table V shows the evaluated processing times using each module's maximum operational frequency. For the complete SFFT core, we run the entire design on a single clock frequency. The critical path of the design lies in the Voter module, specifically in the filter modules that check whether a candidate frequency falls in a high bucket for the corresponding iteration. This check requires mapping the candidate to a bucket index, which is used to select a single value out of the 4096-point vector. We store this vector in Block RAMs, using a vector size of 64 that balances the number of cycles to initialize the filter and the size of the multiplexer that selects the appropriate index out of each 64-point value. This is a significantly better solution than use of Slice Registers, as it reduces the design congestion and allows routing to be completed with desired time constraints.

The steady state throughput of our SFFT Core design is 116,024 clock cycles with a 100 MHz clock, which translates to 1.16 milliseconds per million-point sparse Fourier transform. This design is the first million-point FPGA implementation of the SFFT algorithm. The initial single-threaded software implementation [4] of the algorithm takes 190 milliseconds to complete a transform with the same parameters, while executing on an Intel Core i7-2600 CPU. A recent multi-threaded software implementation [5] takes 100 milliseconds for the same problem size, while executing on an Intel Xeon E5-2660 CPU. Though the CPU-based designs work on floating-point data, our fixed-point FPGA implementation is accurate enough for applications under consideration due to the large number of fractional bits used in the input data type. Our FPGA design is $85\times$ faster than the latter CPU implementation, while having the benefits of operating in a single FPGA form factor and power budget as compared to that of a multi-core CPU.

Table V. Processing times in milliseconds accounting for the maximum operating frequency. The complete design runs on a single frequency.

Design	Steady-state Throughput (cycles)	Maximum frequency (MHz)	Processing Time (ms)
Dense-FFT	54612	121.2	0.45
Max Selector	47104	134.7	0.35
Voter	68816	100.1	0.69
Value Compute	33788	121.9	0.28
SFFT Core	**116024**	**100**	**1.16**

V. Conclusion

In this paper, we have presented the hardware implementation of a million-point sparse FFT design. The design has been parameterized and developed in a modular fashion, enabling its use in a wide variety of sparse FFT applications. This implementation fits within a single Virtex-6 FPGA and can complete the processing of a million-point frequency-sparse input data to generate the indices and values of the 500 most significant frequency coefficients every 1.16 ms. Our design provides a significant speedup over published SFFT software implementations. This high-throughput FPGA implementation of the sparse FFT algorithm allows use of the million-point Fourier transform in mobile and low-power devices for applications dealing with frequency-sparse data.

Acknowledgement

This work was supported by funding from QCRI (Agrmt. Eff. 6/27/12) and MIT-Lincoln Labs (PO7000170673).

References

[1] H. Hassanieh, P. Indyk, D. Katabi, and E. Price, "Simple and practical algorithm for sparse fourier transform," in *Proceedings of 23rd Symposium on Discrete Algorithms, SODA*, pp. 1183–1194, 2012.

[2] H. Hassanieh, P. Indyk, D. Katabi, and E. Price, "Nearly optimal sparse fourier transform," in *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC*, pp. 563–578, 2012.

[3] O. Abari, E. Hamed, H. Hassanieh, A. Agarwal, D. Katabi, A. Chandrakasan, and V. Stojanovic, "A 0.75-million-point fourier-transform chip for frequency-sparse signals," in *Solid-State Circuits Conference, 2014 IEEE International*, pp. 458–459, Feb 2014.

[4] H. Hassanieh, P. Indyk, D. Katabi, and E. Price, "SFFT Sparse Fast Fourier Transform [online]." http://groups.csail.mit.edu/netmit/sFFT/code.html.

[5] J. Schumacher, "High performance Sparse Fast Fourier Transform," Master's thesis, ETH, Zurich, Switzerland, May 2013.

[6] N. Dave, M. Pellauer, S. Gerding, and Arvind, "802.11a transmitter: a case study in microarchitectural exploration," in *MEMOCODE*, pp. 59–68, 2006.

[7] S. He and M. Torkelson, "A New Approach to Pipeline FFT Processor," in *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pp. 766–770, 1996.

[8] A. Ioannou and M. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks," *Networking, IEEE/ACM Transactions on*, vol. 15, no. 2, pp. 450–461, 2007.

[9] Bluespec, Inc., Waltham, MA, *Bluespec SystemVerilog Reference Guide*, September 2013.