# Implementation Study of FFT on Multi-Lane Vector Processors

Bogdan Spinean*, Georgi Gaydadjiev*†
† Department of Computer Science and Engineering, Chalmers University of Technology, Sweden
*Computer Engineering Laboratory, EEMCS, Delft University of Technology, The Netherlands
{b.spinean, g.n.gaydadjiev}@tudelft.nl

***Abstract*—In this paper we extend a custom FFT vector architecture by adding multiple lane capabilities and study its hardware implementation. We use the six step algorithm to segment a long Fourier transform of size $N = Z \times L$ into $L$ smaller transforms of size $Z$. We split the data into pairs of vector registers (for the real and imaginary part), each containing $Z$ elements. A vector register pair with its corresponding functional unit form a single lane replicated $L$ times. While smaller transforms proceed iteratively all of them are computed in parallel. The shorter FFT transforms along the X dimension are computed using previously proposed vector permutations while the transforms along the Y dimension are performed using a simple butterfly network that handles inter-lane communication. All data patterns required by the FFT computation are generated implicitly in hardware by a simple control unit. No data transposition is required and the twiddle factors are stored locally inside the functional units. We validated our design through simulation and ASIC synthesis targeting 90nm CMOS technology. We compare three possible configurations for computing a 256point FFT, all running at 217 MHz with $Z \times L$ equal to: a) $32 \times 8$, b) $16 \times 16$ and c) $8 \times 32$. Configuration a) is the smallest and the slowest; configuration b) requires 1.43 times fewer cycles but 1.64 more area while configuration c) requires 1.76 times fewer cycles and is 3.16 times larger. Unlike other high performance FFT implementations, our design offers the possibility to trade-off execution speed for two resource types: vector register size and number of lanes. Another important contribution is the possibility to execute 2D FFT without any HW modifications or special provisions.**

## I. Introduction

The Discrete Fourier Transform (DFT) has extensive applications in science and engineering ranging from spectral analysis to data compression and partial differential equations. A real breakthrough in the use of the DFT has been brought by the Cooley-Tukey (CT) FFT algorithm [10] that reduced the computation complexity from $O(n^2)$ to $O(n \times log(n))$. The DFT is defined by the formula:

$$X_k = \sum_{n=0}^{n-1} x_n e^{-\frac{2\pi i}{N} nk} \tag{1}$$

The radix 2 decimation in time recursively divides a DFT of size $N$ into two interleaved DFTs of the even and the odd input indexes and then combines the two results to produce the DFT of the overall sequence.

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + e^{-\frac{2\pi i}{N}k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k} \tag{2}$$

$$X_k = E_k + e^{-\frac{2\pi i}{N}k} O_k \tag{3}$$

Due to the periodicity of the DFT, the outputs for $N/2 \leq k < N$ from a DFT of length $N/2$ are identical to the outputs for $0 \leq k < N/2$ and therefore, $E_{k+M/2} = E_k$ and $O_{k+N/2} = O_k$. This way, the large transform can be reconstructed from the smaller ones and the algorithm is accelerated by re-using intermediate results to compute multiple DFT outputs.

$$X_k = \begin{cases} E_k + e^{-\frac{2\pi i}{N}k} O_k & if \;\; k < N/2 \\ E_{k-N/2} + e^{-\frac{2\pi i}{N}(k-N/2)} O_k & if \;\; k \geq N/2 \end{cases} \tag{4}$$

The computation involved is not complicated, each butterfly operation consists of one complex multiplication and two complex additions. The data accesses have a very clear, regular and highly predictable pattern. However, as the computation progresses, the input memory locations are at exponentially increasing distances. For large transform sizes, the data will not fit in superscalar processors' caches, in the GPGPUs' registers or in the registers of vector processors and thus intermediate results must be spilled to main memory [1], [5].

An important improvement in FFT computation performance was brought by the six step algorithm [6]. This algorithm greatly increases parallelism and locality as it divides the FFT into multiple, smaller steps, some truly independent.

In this paper we extend the vector based six step FFT algorithm presented in [2] with multi lane capabilities. Compared to existing FFT implementations, our advantages are twofold. At runtime, we support multiple processing modes, including 2D transforms. Moreover our approach enables additional flexibility at late system design stages since, for a fixed transform size $N = Z \times L$, multiple implementation possibilities exist, each with different vector register sizes $Z$ and number of lanes $L$. Thus, the designer can trade-off speed

for area (and consequently power) allowing a closer fit within the design constraints.

The contributions of this paper are:

- we extend the vector processor design in [2] with multiple lanes capability and with a simple butterfly network to generate necessary inter-lane communication patterns;
- multiple 1D and 2D FFT operations support during runtime and multiple design options with trade-offs between execution speed and area for a given transform size;
- we present an ASIC 90nm implementation that operates at 217MHz clock frequency;
- three configurations are presented for the 256point FFT and we show that reducing the number of cycles by 1.43 costs 1.64 additional area and 1.76 improvement results in 3.16 bigger area.

The remainder of this paper is organized as follows: Section II presents the background in traditional high performance FFT hardware implementations, describes the Vector permutations introduced in [2] and the six step algorithm. Section III presents the proposed architecture describing the hardware blocks and an example. Section IV contains the synthesis results and discusses the design's scalability and finally, Section V presents our conclusions.

## II. Background and Related Work

Traditional high performance custom FFT architectures are based on either pipeline designs or column designs. In the case of pipeline architectures, the building blocks are butterfly processors and shift registers that ensure that data elements arrive at the functional units in the required sequence. The butterfly processors and shift registers are cascaded $log_r N$ times (where $N$ is the transform length and $r$ is the radix) corresponding to the $log_r N$ stages of the computation [7], [14], [12]. Column architectures are composed of $N/r$ processing elements such that for each step all $N/r$ butterflies are computed in parallel. Data shuffling for the column designs is performed by the interconnection network [8], [9]. In both cases the number of functional units is fixed at $log_r N$ for the pipeline implementations and $N/r$ for column architectures.

The proposed architecture fits in neither of these categories and is based on implementing the six step algorithm on vector processors. Compared to other high performance FFT implementations, our advantages are twofold. At runtime, we support multiple processing modes: one long transform, many shorter transforms or 2D transforms. Switching between these mode is done by executing all or only part of the steps presented in the next section. While most custom FFT implementations can be easily adapted for shorter FFT transforms, few of them can perform 2D transforms without additional hardware and control. The second advantage of our approach is the additional flexibility at late system design stages. While the pipelined or column approaches require a fixed number of processing elements ($log_r N$ for the pipeline implementations and $N/r$ for column architectures), our design, for a given transform size $N = Z \times L$, can be implemented in multiple configurations by varying the vector register size $Z$ and the number of lanes $L$. This way, the designer has a lot of freedom in choosing the configuration with the appropriate area and speed and can closely fit within the design constraints while at the same time gain more flexibility for the design of other components of the system. In terms of design flexibility we position in the middle ground between FPGA implementations and traditional high performance ASIC designs.

```
Step 1. transpose
Step 2. N1 independent N2 point FFTs
Step 3. twiddle factor multiplication
Step 4. transpose
Step 5. N2 independent N1 point FFTs
Step 6. transpose
```

Fig. 1. The six step FFT algorithm

**The six step algorithm** sketched in Figure 1 uses a very important property of the FFT that a long transform of length $N = N_1 \times N_2$ can be computed as $N_1$ independent transforms of length $N_2$, followed by the computation of $N_2$ independent transforms of length $N_1$. All intermediate results are then combined together to form the final result. We focus our attention on Steps 2 and 5 of performing independent transforms. We further assume that $N_1 = L$, the number of lanes and that $N_2 = Z$, the length of a vector register. For simplicity we have chosen to use radix 2 functional units. Similar principles apply for higher radixes without loss of generality.

For the original Cooley-Tukey algorithm [10] the results of the butterfly will be stored in the same position as the inputs and thus the algorithm can be performed in place. However other variations, like the Stockham algorithm [15] change the layout of the array at every step and thus additional storage and index calculation are required.

**Vector processors** have been very powerful in areas where high memory bandwidth and regular computation are needed. The first FFT algorithms that have been implemented on vector processors were simple radix-2 algorithms for arrays of lengths $N = 2^p$ [13], such as the Pease [11] or the Stockham [15] algorithms. The algorithms that inspired the six step algorithm were the mixed radix FFTs [3] and then the prime-factor algorithms. Temperton [4] reports on average 97% functional unit utilization when processing multiple transforms in parallel on the Cray-1 vector processor.

The additional parallelism of the six step algorithm is utilized by loop interchange [4]. All details of the FFT indexing are transferred to outer loops and have no impact on vectorization. The fact that the transforms are shorter than the original transform does not have any influence on the vector implementation. The vector register permutations that we have introduced in [2] take advantage of the shorter transforms and greatly reduce the number of memory accesses.

**Vector Register Permutations** extend the state of vector registers with a permutation unit that allows reading and writing the vector registers in a different order. The vector register contains $Z$ elements, $Z = 2^k$. To access those $Z$ elements we require $k$ bits: $x_{k-1}, x_{k-2}, ..., x_1, x_0$. We define a Vector Permutation as a sequence of $k$ numbers as follows:

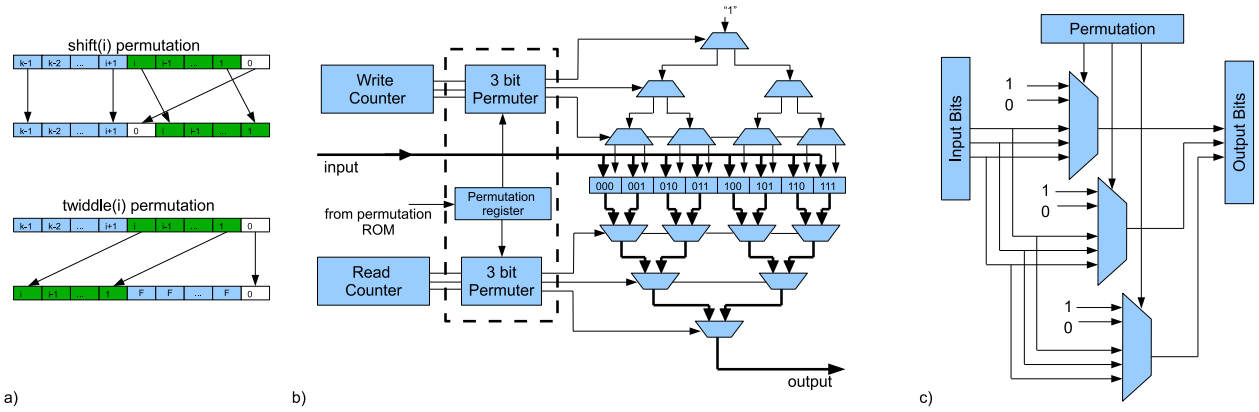


Fig. 2. a) Example permutation (2,0,'1') for the case of Z=8; b) The structure of a vector register extended with the permutation unit; c) The Permutation Unit is implemented using multiplexers.

$(p_{k-1}, p_{k-2}, ..., p_1, p_0)$. For $i = 0$ to $k-1$, bit $p_i$ can take any of the following values: a) either of the input bits $x_{k-1}, x_{k-2}, ..., x_1, x_0$; b) 'T'; c) 'F' with 'T' being a constant value of 1 and 'F' being a constant value of 0.

In [2], we have introduce two permutations that are used to formulate the Cooley-Tukey algorithm in terms of permutations: the $shift(i)$ and $twiddle(i)$ permutations (Figure 2 a)). The $shift(i)$ permutation is used by the data vector registers and is defined as:

```
for j = 1 to k-1 do
  if j > i
    p_j = x_j
  else if j==i
    p_j = x_0
  else
    p_j = x_{j+1}
```

If $Z = 8$, $k = 3$, $shift(1)$ becomes Permutation (2,0,1):

- the $3^{rd}$ output bit becomes the 3rd input bit (index 2);
- the $2^{nd}$ output bit becomes the 1st input bit (index 0);
- the $1^{st}$ output bit becomes the 2nd input bit (index 1);

By applying the (2,0,1) permutation, when reading the vector register, the sequence of elements is:

before permutation: $000, 001, 010, 011, 100, 101, 110, 111$

after permutation: $000, 010, 001, 011, 100, 110, 101, 111$

The vector register holding the twiddle factors uses the $twiddle(i)$ permutation defined as:

```
threshold = k-1-i
p_0 = x_0
for j = 1 to k-1 do
  if j > threshold
     p_j = x_{j-threshold}
  else
     p_j = 'F'
```

## III. Proposed Solution

In this section, we describe the proposed architecture. We start by briefly presenting the steps of the execution, we then discuss in detail the building blocks of our design. In the final part of this section we provide a detailed example of all thee steps required for FFT computation. Figure 3 shows the block

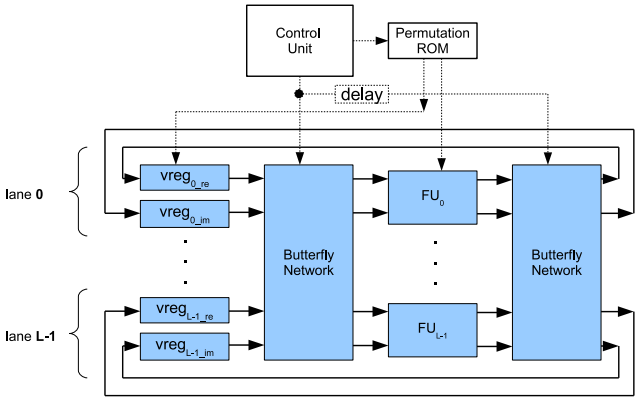


Fig. 3. The multi lane FFT architecture consists of $L$ lanes, two butterfly network and the control unit. A lane contains two vector registers and a functional unit.

scheme of our design. We process an $N = Z \times L$ point FFT transform by first computing $L$ independent transforms of size $Z$. We call this the "X FFT step". We then perform what we call the "correction step" by multiplying the intermediate results by correction factors. The third step is the "Y FFT step" that consists of $Z$ independent transforms of size $L$.

We split the data into pairs of vector registers (for the real and imaginary part), containing $Z$ elements. The vector register pair with its corresponding functional unit form a lane and we replicate this design $L$ times. Each lane, using vector permutations, can perform a transform of size $Z$ in $log(Z)$ iterative steps, each step completing in a number of cycles in the order of $O(Z)$ [2]. All lanes operate in lock step, using the same vector permutations. Therefore the "X FFT step" will execute in $O(Z \times log(Z))$ cycles.

The "correction step" consists of multiplying each element by a twiddle factor. The functional units contain the necessary twiddle factor, therefore for this step, in each lane, the contents of the vector registers will be multiplied by the correction twiddle factors stored locally in the lane's functional unit.

The first two steps required no inter-lane communication. However, the "Y FFT step" is made possible by the use of the butterfly interconnection network. By selecting the proper

network configurations, two elements from different lanes are brought together at a functional unit, the butterfly is calculated and the results are written back to the locations where the inputs originated. To the vector registers and the functional units the "Y FFT step" is identical to the "X FFT step". The only thing that changes is the network switching. Therefore, the duration of the "Y FFT step" is in $O(Z \times log(L))$ cycles.

For clarity we distingwish between steps and iterations. We call the "X FFT" and "Y FFT" "steps" and each step consists of $log(Z)$ and respectively $log(L)$ "iterations".

### A. The Hardware Structure

*1) The Vector Registers:* Each vector register contains $Z$ elements that are 32bits wide (see Figure 2). For our implementation, the reading and writing of the $Z$ elements is an atomic operation that takes $Z$ consecutive cycles without the possibility to stop the data transfer in between. The addition of the vector permutations described in [2] allows reading or writing the elements in various orders such that at the output of the vector register the two elements required for a butterfly come at consecutive positions. For example in the first iteration of the "X FFT" step the butterflies are performed between consecutive elements. Therefore, the vector registers are read sequentially, the permutation used is the identity permutation. For the second iteration of the same step, the butterflies are performed with elements that have indexes in the sequence (0,2) for the first butterfly, (1,3) for the second butterfly and so on (see figure 6). The vector registers can be configured with the correct permutations such that the elements will be read in the correct sequence. Each lane has two vector registers, one for the real and one for the imaginary data elements.

*2) The Functional Units:* The functional units (FUs) are specialized for doing the butterfly computations. The control unit through the use of vector permutations or through the butterfly interconnection network ensures that the two complex inputs for the butterflies $a$ and $b$ arrive at the functional unit in two consecutive cycles. The control unit also sets the permutation to the twiddle vector register. The FUs contain two sets of twiddle factors. All the twiddle factors required by transforms of length $Z$ or $L$ ($L < Z$) are in the vector register named "twiddle". These are the $Z/2$ order roots of unity. In iteration one, only one twiddle factor will be used. This is obtained by setting a permutation that generates at the output the same vector register element every time. The second iteration will require two twiddle factors and this is obtained by setting a permutation that will alternate between the two required vector elements. The second set of twiddle factors is contained in the block named "correction". These are $Z$ of the $N/2$ order roots of unity, and are used for the correction step. The correction calculation consists of a single complex multiplication and all input data is multiplied by the corresponding correction twiddle factor. No permutations are required and thus the correction twiddle factors are stored in a shift register instead of a vector register. The twiddle block is identical for all lanes and always uses the same permutation in all lanes. The correction block is unique for each lane, the $L$ lanes contain all the $N/2$ order roots of unity. Since the same twiddle factors are used for any input data, we have chosen to store them inside the functional units. This brings a double benefit. First, there is no bandwidth required to transfer the twiddle factors and second, because the twiddle factors are physically located right next to the complex multiplier this translates into shorter wires and the design is more efficient.

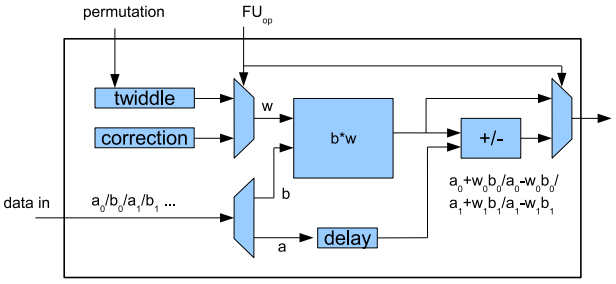


Fig. 4. The butterfly FU performs either butterflies between two consecutive inputs or complex multiplication between the inputs and the correction factors.

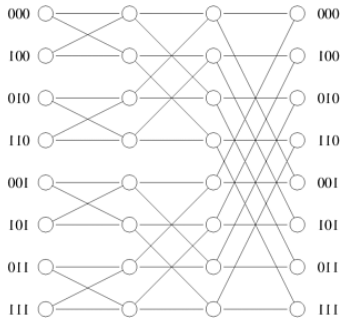


Fig. 5. The butterfly Network has $2 \times L$ inputs and outputs and it consists of $1 + log(L)$ steps of 2x2 switches.

*3) The Communication Network:* The Butterfly Network, used for inter-lane communication is shown in Figure 5. The basic building block is the 2x2 switch with two possible configurations: outputs are same as inputs or outputs swap position. We use a network with $2 \times L$ inputs and outputs (we have $2 \times L$ vector registers for the real and imaginary part of the data). Therefore it has $1+log(L)$ levels, each consisting of $L$ 2x2 switches. The butterfly network is not blocking, in each cycle the $2 \times L$ outputs are connected to all the inputs. Each input is connected to a single output, no two inputs can connect to the same output and no input can connect to more than one output. The butterfly network does not provide all possible input-output connections but all of the patterns required by the FFT computation are supported.

Our design requires two butterfly networks one for passing data from the vector registers to the functional units and the other one from the functional units back to the vector register. The intermediate FFTs are computed in place, and therefore the results of the butterflies will be stored in the same position as the inputs. Thus, the second butterfly networks will have the the same configuration as the first but delayed by an amount of clock cycles equal to the functional unit latency.

*4) The control unit:* The control unit is a hierarchical FSM that executes (besides the basic I/O) the following three

operations: a) X FFT, b) Y FFT and c) correction step. By combining these operations we gain the flexibility of executing multiple types of FFT computations. A 1D FFT of size $N = Z \times L$ can be executed by performing the operations in the order: X FFT, correction, Y FFT. A 2D FFT of the same size can be computed by performing only operations a) and b), without the correction step. A third possible command is the computation of $L$ independent transforms of size $Z$. In this case only step a) X FFT is required.

### B. Motvational Example

In this subsection we discuss in detail how a 32point 1D FFT is computed by using $L = 4$ lanes, each holding $Z = 8$ elements. We have chosen these numbers because they are small enough that the algorithm execution can be presented entirely and also because they are large enough to give an idea of the characteristics of the proposed architecture.

Using the 6 step algorithm we segment the 32 data elements into a logical array of $8 \times 4$ that maps to 4 lanes containing 8 elements. The first step is what we call the **"X FFT step"** and consists of computing 4 independent 8point transforms, one in each lane. Figure 6 shows the $(log(Z) = 3)$ iterations of the 8 point FFT transform. For this step there is no inter-lane communication and the butterfly networks are configured such that the output port is the same as the input port.

The butterfly computation produces 2 complex outputs $a + (b * w)$ and $a - (b * w)$ and requires 3 complex inputs: the two data elements ($a$ and $b$) and the twiddle factor $w$. The two data elements are stored in two vector registers, one for the real part, another for the imaginary part. The twiddle factors are stored in a vector register inside the functional unit and contains alternating real and imaginary parts. The number of twiddle factors is equal to the butterfly computations required which in our case is $Z/2$.

Our goal is to read these registers in a suitable order such that at the functional units data will arrive grouped per butterfly: the first two cycles the FU will receive the two data elements and the twiddle factor required for the first butterfly, the following two cycles the data elements and twiddle factor for the second butterfly and so on. The pipelined functional unit will generate the two results of the butterflies in consecutive cycles that must be written back to the vector register in the same order as the data elements have been read.

We do not consider the data values as they do not have any impact on the techniques described here. Instead, we shall focus on the elements' position. Table I presents the sequences of addresses required by the FFT algorithm presented in Figure 6. The four pairs of columns represent the butterflies B0, B1, B2 and B3 computed during each iteration. Groups of the rows correspond to the iterations of the FFT algorithm. For each iteration, we must determine one Permutation for the data array and one Permutation for the twiddle factors.

For **the first iteration**, in the vector registers, the data array is in the required order, butterflies are computed between elements of consecutive positions. Therefore, we configure the two data registers with the identity permutation (2,1,0). We

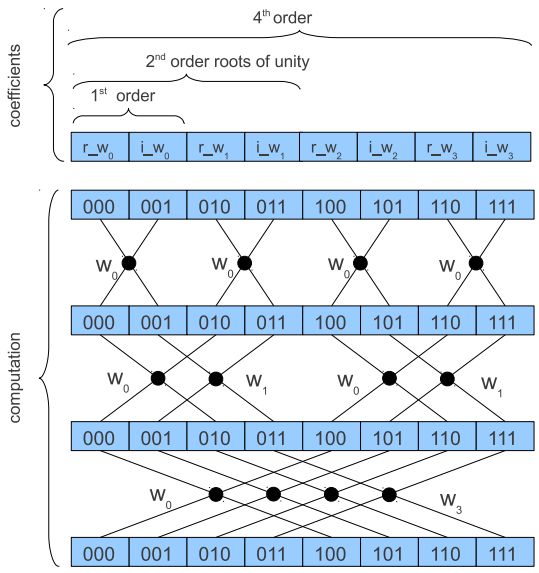


Fig. 6. The butterfly operations of an 8 point Cooley Tukey FFT.

| Permutation | | 000 001 B0 | | 010 011 B1 | | 100 101 B2 | | 110 111 B3 | |
|---|---|---|---|---|---|---|---|---|---|
| Step 1 | data | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| (2,1,0) | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| | twiddle | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| (F,F,0) | | 000 | 001 | 000 | 001 | 000 | 001 | 000 | 001 |
| Step 2 | data | 0 | 2 | 1 | 3 | 4 | 6 | 5 | 7 |
| (2,0,1) | | 000 | 010 | 001 | 011 | 100 | 110 | 101 | 111 |
| | twiddle | 0 | 1 | 4 | 5 | 0 | 1 | 4 | 5 |
| (1,F,0) | | 000 | 001 | 100 | 101 | 000 | 001 | 100 | 101 |
| Step 3 | data | 0 | 4 | 1 | 5 | 2 | 6 | 3 | 7 |
| (0,2,1) | | 000 | 100 | 001 | 101 | 010 | 110 | 011 | 111 |
| | twiddle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| (2,1,0) | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

TABLE I
THE ADDRESS SEQUENCES REQUIRED BY THE 8 POINT FFT USING THE COOLEY-TUKEY ALGORITHM.

can follow in Table I that the first butterfly (B0) is computed between element 0 and element 1, the second butterfly (B1) is computed between element 2 and element 3 and so on.

The first iteration requires only one twiddle factor for all butterflies that must be replicated in all the positions of the corresponding vector register. Since the twiddle factor is a complex number, the output of the vector register must alternate between the first two elements of the vector array: the element on position 0 (the real part), followed by element on position 1 (the imaginary part), followed again by element 0, then element 1 etc. In order to obtain the required sequence of addresses we need the following permutation word: the first two bits must stay constant at $F$ while the LSB must toggle every cycle. The permutation for the twiddle factors is (F,F,0);

For **the second iteration** the first butterfly B0 is computed between elements 0 and 2, B1 is computed between elements 1 and 3 and so on. By examining the required sequence of addresses, we notice that the second bit of the address

swaps first, then the LSB and then the MSB. Therefore the Permutation becomes (2,0,1).

The twiddle factors required in the second iteration are the second order roots of unity which are on position 0 and position 4. Thus, the permutation required to read the vector register containing the twiddle factors is (1,F,0).

For **the third iteration**, following the same reasoning, we obtain the permutation for the data array to be (0,2,1) and the permutation for the twiddle factors to be (2,1,0).

**The second step** in our 32point 1D FFT computation is the "correction step". The data from each vector register is multiplied by the correction factors stored inside the functional units and written back to the vector registers. For this steps the vector registers are configured with the identical permutation, they are read and written in the normal sequential order. The butterfly networks are configured to their default state, the outputs are equal to the inputs.

**The third step** is the "Y FFT step". In this step $Z = 8$ independent transforms of length $L = 4$ are computed in parallel. This third step consists of $log(L) = 2$ iterations that are displayed schematically in Figure 7. The central role is played by the butterfly networks.

The first of the 8 transforms has its elements on the first position in the vector registers of each lane (0, 8, 10, 18). The second one has its elements on the second position of the vector registers and so on. Let us focus on the first transform. In the initial iteration the butterflies are computed between elements (0, 8) and (10, 18) while in the second iteration the butterflies are computed between elements (1, 10) and (8, 18). The data must reach the functional units grouped per butterflies in consecutive cycles. Therefore we must configure the system such that elements (0 and 8) reach a FU in consecutive cycles and the same for (10 and 18). The first observation is that lanes $L_1$ and $L_3$ must be delayed by one cycle. We configure the butterfly network such that in the even cycles the butterfly network does not cross at all while in the odd cycles the first level of $2 \times 2$ switches cross. We also configure the vector registers with the identical permutation, data is read sequentially, in the natural order. The twiddle factors in the FUs use the same permutations as in the "X FFT step", for this initial iteration the permutation is $twiddle(0)$.

The result is that the elements on even positions in $L_0$ get to $FU_0$ while those in odd positions arrive at $FU_1$. The data elements of $L_0$ and $L_1$ are interleaved, the even ones in both $L_0$ and in $L_1$ arrive at $FU_0$ while the odd elements in both $L_0$ and in $L_1$ arrive at $FU_1$. The data elements arrive in the correct order to the $FUs$. Summing up, during the first iteration the two butterflies of the 1st transform are computed in $FU_0$ and $FU_2$ respectively, while the two butterflies of the 2nd transform are computed in $FU_1$ and $FU_3$ respectively. The butterfly network that connects the $FUs$ back to the vector registers has the same alternating configuration but it is delayed by a number of cycles equal to the $FUs$ latency.

During **the second iteration** vector registers in lanes $L_2$ and $L_3$ are read one cycle later. During the even cycles the butterfly network does not cross at all, while during the odd cycles

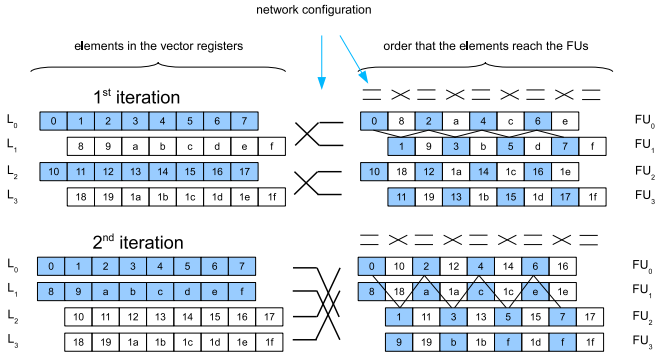


Fig. 7. During The Y FFT step of the FFT computation each iteration half the registers are read one cycle later and one single level of the butterfly network switches every cycle.

the second level of switches cross. The data vector registers are read sequentially, while the twiddle vector registers inside the FUs are configured with the $twiddle(1)$ permutation. The result is very similar to the previous iteration, the elements on even positions in both $L_0$ and $L_2$ get to $FU_0$ while those in odd positions arrive at $FU_2$. The data elements of $L_0$ and $L_2$ are interleaved, the even ones in both $L_0$ and in $L_2$ arrive at $FU_0$ while the odd elements arrive at $FU_2$.

We can extend the working prinicple for any value of $L$. During iteration $i$ the elements of lanes $L_0$ and $L_{2^i}$ are interleaved, the even ones arrive at $FU_0$ and odd at $FU_{2^i}$.

## IV. Experimental Results

### A. Synthesis results

We have designed the multi-lane FFT implementation in behavioural Verilog. The emphasis was on making the design as generic as possible in order to be implementation independent. We have synthesised our design in 90nm commercial standard cell technology using Cadence Encounter RTL Compiler. We have run the steps up to placement. For all configurations of $Z$ and $L$ the frequency of our design is $217MHz$, the critical path is in the functional units and can be improved by using deeper functional unit pipelining.

In terms of area, the three major components of our design are: the functional units (including the twiddle factors), the butterfly networks and the vector registers containing the data. The control unit represents in all the studied configurations less than 1% of the total area. Figure 8 shows the area distribution when varying both $Z$ and $L$ from 4 to 32 in powers of two.

Most of the area is occupied by the functional units. For small values of $Z$ and $L$ the functional units occupy around 70% of the design's area. For the largest designs this percentage falls down to 40% but still remains dominant component in terms of area. The area taken by the butterfly networks fluctuates greatly. For designs with 4 lanes the butterfly network requires around 10% of the total area while for 32 lanes this percentage goes up to 25-30%. This is because more lanes translate into more levels in the butterfly network and also, since the butterfly network is dominated by wires, the larger the network the higher the overhead. While on average, for the entire design the wires take 30 to 45% of the total

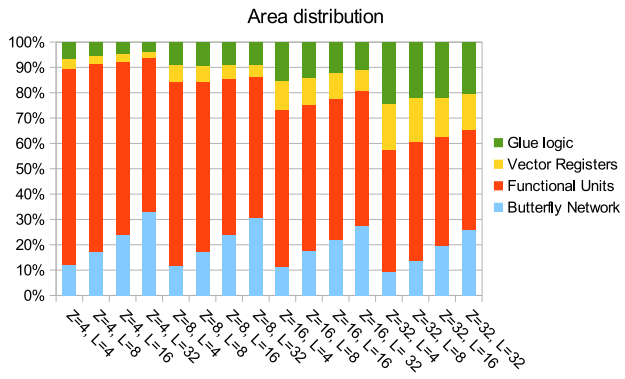


Fig. 8. The area occupied by the major components of the design for both $Z$ and $L$ ranging from 4 to 32 in powers of two.

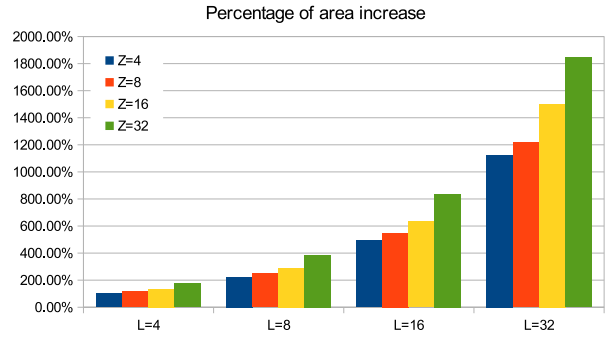


Fig. 9. The increase in the design area when both $Z$ and $L$ ranging from 4 to 32 in powers of two. The baseline design is for $Z = L = 4$

area, the butterfly networks consist mostly of wires. For the smallest configurations of $Z = L = 4$ the net area is 66% of the butterfly network's area while for the largest configuration of $Z = L = 32$ the wires account for 80% the the butterfly network's area. The vector registers holding the data have the smallest impact with regards to the overall area.

### *B. Register size and number of lanes scalability*

In the following paragraphs we discuss the effects of changing the design parameters. We investigate the changes in total area, the modifications to the control unit, the butterfly networks and the vector registers. We also study the total execution time, in terms of cycles. Figure 8 compares the total area of the studied configurations. It is essential to note that the number of lanes $L$ and the vector register size $Z$ can only be powers of two. This is because the FFT algorithm requires logarithmic number of steps and also the butterfly network requires a logarithmic number of levels. Another important note is that the vector register size $Z$ should close to the number of lanes $L$. Otherwise the design becomes unnecessarily unbalanced.

Doubling the number of lanes $L$ (see Figure 3) brings no conceptual change in the FSM. The "Y FFT step" will contain an additional iteration. However, the additional twiddle factors are already in the functional units. There is no change whatsoever in the structure of a lane: the vector registers and

| Total number of cycles | Z=4 | Z=8 | Z=16 | Z=32 |
|---|---|---|---|---|
| L=4 | 84 | 126 | 204 | 362 |
| L=8 | 112 | 160 | 248 | 424 |
| L=16 | 144 | 198 | 296 | 490 |
| L=32 | 180 | 240 | 348 | 560 |

TABLE II
FOR EACH CONFIGURATION, THE NUMBER OF CYCLES REQUIRED TO COMPLETE THE MAXIMUM SIZE FFT.

the functional units remain unchanged. The butterfly networks require an additional level and the butterfly network area more than doubles since it is dominated by wiring. When doubling $L$, the main effect is that the total design area more than doubles. This is because we double the number of functional units and vector registers and because the butterfly networks greatly increase in size (see Figure 8).

Doubling the vector register size $Z$ brings no conceptual change in the FSM. The "X FFT step" will contain an additional iteration and will require double the number of twiddle factors stored in the functional units. The structure of the butterfly network does not change at all. For small designs, doubling the vector registers size increases total area by a fraction, around 10% or less. For large designs, however, since the wiring becomes more important, doubling the vector register size increases the area by a substantial amount.

Table II presents the number of cycles required for each configuration to compute a 1D FFT of size $Z \times L$. These numbers have been obtained through simulation. As a rule of thumb the duration of the "X FFT step" is $D_X = log(Z) \times (Z + 2 \times log(L) + lat)$. There are $log(Z)$ iterations, each consisting of reading the vector registers ($Z$) + the functional unit latency ($lat$) + the latency of passing twice through the $log(L)$ levels of the butterfly network. The approximate duration of the correction step is $D_{corr} = Z+2\times log(L)+lat)$. That is, the duration of a single iteration of the "X FFT step". The approximate duration of the "Y FFT step" is $D_Y = log(L) \times (Z + 2 \times log(L) + lat)$, $log(L)$ iterations, each of the same length as the "X FFT step" iteration.

Let us focus on the 1D FFT that is 256 elements long. There are three possible configurations of our design: configuration a) $Z \times L = 32 \times 8$; configuration b) $Z \times L = 16 \times 16$; and configuration c) $Z \times L = 8 \times 32$. We will compare these three configurations by following Table II and Figure 9. For this comparison, let us take the baseline as configuration c) which is the slowest in terms of number of cycles and the smallest in terms of area. Then, configuration b) is 1.43 times faster and 1.64 times larger (in terms of cycles and area) while configuration c) is 1.76 times faster and 3.16 times larger.

We thus conclude that we trade-off execution speed for two different resource types: vector register size and number of lanes. Depending on the implementation constraints, the user may choose between multiple implementations. General guidelines for increasing performance and scaling up the design are as follow: if the design is area constrained than increase $Z$. If the design is constrained by execution time, then increase $L$. If efficiency is the main concern, a balanced design (where $Z = L$) offers the best trade-off between area resources

and speed. On the opposite, the most inefficient configurations are those where $Z << L$ or $Z >> L$.

For **higher radixes** the concept remains the same. The only block that changes is the FU block. We use radix 2, with the two inputs $a$ and $b$ arriving at the FUs in two consecutive cycles while radix 4, requres four inputs and outputs that stream through the FUs in consecutive cycles. This is compatible with the concept presented in this paper. For higher radixes we need less iterations but larger functional units, with more multipliers.

## V. Conclusions

We have extended an architecture for FFT computation described in [2] with multi-lane capabilities. We use the six-step algorithm to segment a long transform of size $N$ into $Z$ blocks of length $L$, where $N = Z \times L$. We then map the $N$ complex elements into $L$ pairs of vector registers of size $Z$. The shorter FFT transforms along the X dimension are computed using previously proposed vector permutations that allow the reading and writing of vector register elements using different patterns required by the FFT computation. The transforms along the Y dimension are performed using a simple butterfly network that handles inter-lane communication. The functional units are specialized for computing the radix 2 butterflies and require that the two elements arrive in two consecutive cycles. By using the correct permutation for the data registers and correct switching of the network all the data patterns required by the FFT computation are generated implicitly in hardware.

Compared to state of the art high performance FFT implementations our advantages are two fold. At runtime, we support multiple processing modes: one long FFT transform, many shorter transforms or 2D FFT transforms. The second advantage of our approach is the added flexibility at late design stages of the system that uses the FFT computation. While the pipelined or column approaches require a fixed number of processing elements ($log_r N$ for the pipeline implementations and $N/r$ for column architectures), our design, for a fixed transform size $N = Z \times L$, can be implemented in multiple possible configurations by varying the vector register size $Z$ and the number of lanes $L$. This way the FFT block can fit better within the area or power budget constraints.

We have explored various design sizes raging from $4 \times 4$ up to $32 \times 32$. Our design trades-off execution speed for two types of area resources: vector register size and number of lanes. General guidelines for increasing performance and scaling up the design are as follow: if the design is area constrained then $Z$ should be increased. If the design is constrained by execution time, then increasing $L$ is preferable. If efficiency is the main concern, a balanced design (where $Z = L$) offers the best trade-off between area resources and speed. On the opposite, the most inefficient configurations are those where $Z << L$ or $Z >> L$. As an example we show three possible configurations for computing a 256point FFT, all running at 217 MHz. Compared to the smallest configuration, one of the other three requires 1.43 fewer cycles and requires 1.64 more area while the third one requires 1.76 times fewer cycles and is 3.16 times larger.

## References


[1] A. Nukada, Y. Ogata, T. Endo, S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC.*, pages 1–11, 2008.

[2] B. Spinean, G. Kuzmanov, G. Gaydadjiev. Vector Processor Customization for FFT. In *International Conference on Embeded Computer Systems (SAMOS)*, pages 110–117, 2011.

[3] C. Temperton. Fast Fourier Transforms and Poisson solvers on Cray-1. In *Infotech State of the Art Report: Supercomputers, vol 2*, pages 359 – 379, 1979.

[4] C. Temperton. Implementation of a prime factor FFT algorithm on Cray-1. In *Parallel Computing 6*, pages 99 – 108, 1988.

[5] D Takahashi. An implementation of parallel 1-D FFT using SSE3 instructions on dual-core processors. In *PARA'06 Proceedings of the 8th international conference on Applied parallel computing*, pages 1178–1187, 2007.

[6] D.H. Bailey. FFT's in external or hierarchical memory. In *Journal of Supercomputing 4*, pages 23 – 35, 1990.

[7] E. Bidet, D. Castelain, C. Joanblanq, P. Senn. A fast single-chip implementation of 8192 complex point FFT . In *Journal on Solid-State Circuits*, pages 300–305, 1995.

[8] G. Sunada, J. Jin, M. Berzins, T. Chen. COBRA: an 1.2 million transistor expandable column FFT chip. In *International Conference on Computer Design: VLSI in Computers and Processors*, pages 546–550, 1994.

[9] J. O'Brien, J. Mather, B. Holland. A 200 MIPS single-chip 1k FFT processor. In *International Solid-State Circuits Conference*, pages 166–167, 1989.

[10] J.W. Cooley, J.W. Tukey. An algorithm for the machine computation of the complex Fourier series. In *Mathematics of Computation 19*, pages 297–301, 1965.

[11] M.C Pease. An adaptation of the fast Fourier Transform for parallel processing. In *J. ACM 15*, pages 252–264, 1968.

[12] P. A. Ruetz, M. M. Cai. A real time FFT chip set: architectural issues. In *10th International Conference on Pattern Recognition*, pages 385–388, 1990.

[13] P. N. Swarztrauber. FFT algorithms for vector computers. In *Parallel Computing 1*, pages 45 – 63, 1984.

[14] S. He, M. Torkelson. Design and implementation of a 1024-point pipeline FFT processor. In *Custom Integrated Circuits Conference*, pages 131–134, 1998.

[15] W.T. Cochran, J.W. Cooley, D.L. Favin, H.D. Helms, R.A. Kaenel, W.W. Lang, G.C. Maling, Jr., D.E. Nelson, C.M. Reader and P.D. Welch. What is the fast Fourier transform? In *IEEE, Trans. Audio Electroacoustics Au-15*, pages 45 – 55, 1967.