Vector Processor Customization for FFT

Bogdan Spinean, Georgi Kuzmanov, Georgi Gaydadjiev
Computer Engineering Laboratory, Faculty EEMCS,
Delft University of Technology, The Netherlands
{b.spinean, g.k.kuzmanov, g.n.gaydadjiev}@tudelft.nl

***Abstract*—Processors and memory systems suffer from a growing performance gap between them. Each technology generation increases the on-chip performance capabilities however, memory bandwidth increases at a much slower pace. Therefore, overall performance improvements are constrained by the available memory bandwidth. In this paper, we address the memory bandwidth problem of vector processors by introducing hardware customizations which drastically reduce the memory transfers required by the FFT computation. We show that an FFT transform of length equal to the machine size Z can be performed using only $O(Z)$ memory accesses, hence we reduce the memory bandwidth requirement by an order of $O(log(Z))$ compared to a conventional vector machine. We achieve bandwidth reduction by extending a classic IBM S/370 vector architecture for better register re-use. Our hardware extension completely eliminates the input bit reversal phase of the Cooley-Tukey FFT algorithm. Synthesis results suggest that our extension does not impact the machine cycle time and has a small hardware area overhead of the vector register file of under 4.5% while potentially improving vector performance by a factor of 7.5 for $Z = 256$.**

I. Introduction

The Discrete Fourier Transform (DFT) is widely used 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 FFT algorithm [6] that reduces the computation complexity from $O(n^2)$ to $O(n \times log(n))$.

In this paper, we address FFT implementations of the six step algorithm [3] characterized by increased parallelism and improved locality on three platforms: vector processors, superscalar processors with SIMD support and GPGPUs. We compare the three platforms in terms of their FFT performance theoretical limits: number of memory accesses, the number of overhead instructions executed and the instruction bandwidth.

Our analysis suggests that, of the three considered architectures, vector processors are the most suitable for FFT computation. The main disadvantage of vector processors is the very high memory bandwidth requirements. Therefore, we propose an architectural extension to vector processors that reduces the required memory bandwidth required for FFT computation. We compute the entire FFT transform of number of points equal to the machine vector register length Z avoiding storage of intermediate results in memory. We are thus able to perform the FFT transform using $O(log(Z))$ times less memory bandwidth compared to traditional vector processor implementations.

Since contemporary designs are more constrained by the memory bandwidth rather than by the computational resources, we argue that for a given bandwidth we can potentially improve the overall system performance by allocating more computational resources to utilize the saved bandwidth.

We compare our architectural extension to a baseline vector machine using two methods. Using analytical modeling, we estimate the execution time, the number of memory accesses, the number of address calculation instructions and the number of loop overhead instructions in terms of the machine vector length Z for both cases. We show that asymptotic behavior of both designs is the same and discuss the results for practical values of Z. Through hardware synthesis we show that our extension does not increase the processor's cycle time and that the area overhead is at most 4.5% of the vector register file area. The specific contributions of this paper are:

- an architectural extension of vector processors that allows more efficient register re-use for FFT;
- a micro-architectural implementation of the proposed architectural extension;
- bit-reversed input ordering folded into the computation at no additional cost;
- reduced memory bandwidth requirement by a factor of $log(Z)$ because of better register re-use;

The remainder of this paper is organized as follows: Section II briefly describes the IBM System/370 Vector Architecture highlighting vector processors' key features. Three FFT implementations are then presented and compared: on vector processors, on susperscalar processors with SIMD support and on GPGPUs. Section III presents our architectural extension and describes in detail the Cooley-Tukey algorithm implementation using our approach. Section IV compares the two vector implementation approaches and, finally, Section V presents our conclusions.

II. Background and Motivation

We first provide a short description of the IBM System/370 Vector Architecture [2], [5]. This is the architecture that we, without losing generality, will use as the baseline for our experiments. The machine has 16 vector registers, each containing Z 32bits elements. The machine vector length Z is implementation dependent and equals powers of 2 between 16 to 512 elements. The key insight is that typical vector registers are an order of magnitude larger than typical SIMD registers.

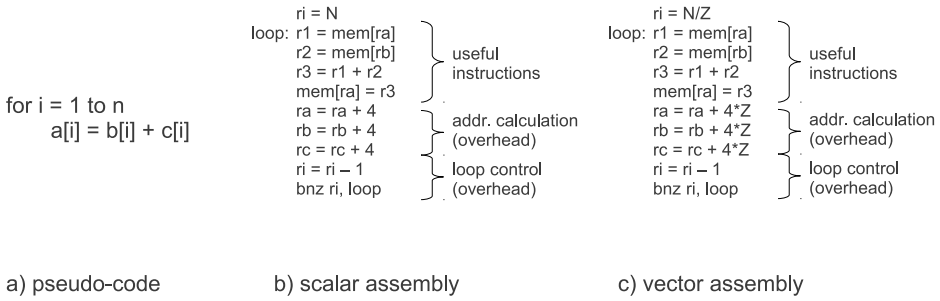

Fig. 1. Scalar code requires Z times more overhead instructions than vector code.

```
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. 2. The six step FFT algorithm.

The Z operations corresponding to the elements of a vector register are computed by a single pipelined 32bit wide functional unit. In each cycle it receives a pair of inputs and generates one result. The elements of the vector registers are read sequentially and they are also written in linear order.

Another key feature of vector processors is that part of index calculations are done implicitly by hardware and also part of the loop control overhead is eliminated. Figure 1 a) shows the pseudo-code of a simple loop which is then translated into assembly instructions for b) scalar machine and c) vector machine. The difference between Figure 1 b) and c) is the total amount of iterations executed. The vector executes Z times fewer index calculation and loop control instructions.

For our estimations, we assume the presence of a vector instruction that performs the butterfly computations. This type of instruction can be a compound instruction used by the IBM System/370 [2]. We focus on the access patterns and simplify our view of the data computation since it is very well understood and documented.

In the following paragraphs we discuss FFT implementations of the six step algorithm [3] on three platforms: vector processors, superscalar processors with SIMD support and GPGPUs. We will perform a comparison in terms of the number of memory accesses, the number of overhead operations executed and the instruction bandwidth. Since the comparison is not straight forward we use the big O notation in our study. For each of these architectures we shall briefly describe how the two benefits of the six step algorithm are utilized: improved locality and additional parallelism available.

The six step algorithm sketched in Figure 2 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 be combined together to form the final transform. We focus our attention on Steps 2 and 5 of performing independent transforms. We further assume that $N_1 = N_2 = Z$ and that

```
for i = 0 to Z do
  for j = 0 to log(Z) do
    for k = 0 to Z/2 do
      calculate addr of a = f1(j,k)
      calculate addr of b = f2(j,k)
      calculate addr of w = f3(j,k)
      load a
      load b
      load w
      (a',b') = butterfly(a,b,w)
      calculate addr of a' = f4(j,k)
      calculate addr of b' = f5(j,k)
      store a'
      store b'
    end
  end
end
```

Fig. 3. Computing Z independent FFT transforms of size Z.

we are using a radix 2 implementation (for higher radix the same principles can be applied) resulting in the pseudo-code shown on Figure 3.

For the original Cookey-Tukey algorithm [6] the results of the butterfly will be stored at the same position as the inputs and thus the algorithm can be performed in place. However other variations, like the Stockham algorithm [16] change the array layout at every step and thus additional storage and index calculations are required. Also, reading the inputs using one access pattern and writing the outputs in another pattern can effectively combine the transposition steps (Steps 1, 3 and 5) into the computation steps (Steps 2 and 4) [1], [12]. It is important to note that in the code of Figure 3 the innermost loop performs the $Z/2$ butterflies required for step j of processing the i^{th} independent transform.

FFT implementations on vector processors. The first FFT algorithms that have been implemented on vector processors were simple radix-2 algorithms for arrays of lengths $N = 2^p$ [11] such as the Pease [10] or the Stockham [16] algorithms. Fixed geometry algorithms [10] can be efficiently vectorized, and they require the same order of memory transfers as the six step algorithm therefore we can safely focus on the latter. The algorithms that inspired the six step algorithm were the mixed radix FFTs [14] and then the prime-factor algorithms. Temperton [15] reports on average 97% functional unit utilization when processing multiple parallel transforms on the Cray-1 vector processor.

The additional parallelism of the six step algorithm is

```
for j = 0 to log(Z) do
  for k = 0 to Z/2 do
    calculate addr of a = f1(j,k)
    calculate addr of b = f2(j,k)
    calculate addr of w = f3(j,k)
    calculate addr of a' = f4(j,k)
    calculate addr of b' = f5(j,k)
    load w
    for i = 0 to Z do
      load a
      load b
      (a',b') = butterfly(a,b,w)
      store a'
      store b'
    end
  end
end
```

Fig. 4. Adaptation of the compuitation of Z independent FFT transforms of size Z for vector processors

utilized by interchanging the inner most loop with the outer loop [15]. All details of the FFT indexing are thus transferred to outer loops and have no impact on vectorization (see Figure 4). Figure 5 shows how the innermost loop in Figure 4 computes butterfly k of step j for all the Z independent transforms. All these butterflies require the same twiddle factor thus, its loading can be done outside the inner loop. Note that elements of a from consecutive iterations will be at constant offsets and thus strided memory accesses can be used for both loading and storing the data. The inner loop in fact loads columns of the $x(N_1, N_2)$ array and computes the butterflies between columns.

Because we have chosen the transform size equal to the machine vector length Z the inner loop can be completely vectorized. It will not require any loop instructions and therefore the address generation is moved from software to the vector implicit address generation. Thus, the loop control and index calculation must be performed only for the outer loops. The total number of such instructions is in the order of $O(Z \times log(Z))$. Since the inner loop is completely vectorized, the total number of dynamic instructions is in the same order of $O(Z \times log(Z))$. The loading and storing of the data elements is performed inside the inner loop and thus the number of memory operations is in the order of $O(Z^2 \times log(Z))$.

SIMD implementations of FFT on cache based machines: Takahashi [12] implemented an FFT algorithm on Intel Xeon processors using SSE3 instructions. The 128bit SSE3 registers are used to store one complex number in double precision floating point. The SIMD capabilities are used for the complex arithmetic rather than butterfly parallelism. The authors use the six step FFT algorithm [3] and exploit the shorter FFT size by using a block size that would fit in the L1 cache. The additional parallelism is used by performing the FFT transforms on multiple cores. From the code snipped described in [12] we observe that for the inner loop of the algorithm 7 out of the 14 operations are used for address calculation. By analyzing the algorithm that the authors have used we can determine that the number of memory transfers, the amount of computation required, the number of address calculation (overhead) instructions and the total number of executed instructions are together in the order of $O(Z^2 \times log(Z))$. This confirms the findings of [9] who reports that after including prefetching, performance of SIMD applications is limited by issue and fetch bandwidth.

Not using the six step algorithm but still relevant for our discussion is the work of Nadehara [7] that implemented a radix-4 decimation in frequency FFT algorithm using 64bit SIMD registers, with the data size being 16bit. The SIMD registers are used to compute 4 butterflies in parallel. While the used data structure allows for easy loading of the data arrays in the SIMD registers, the twiddle factors require additional packing and permutation instructions to place them in the correct positions of the SIMD registers. The authors report using 19 instructions per butterfly, more than 40% of those are used for index and address calculation, SIMD packing, unpacking and are basically overhead instructions.

Talla reported in [4] that in case of SIMD acceleration of Multimedia Applications, the supporting/overhead related instructions dominate media instruction streams accounting for 75-85% of the dynamic instructions.

FFT implementations on GPGPUs. In [1] the authors have implemented a 3D FFT on Nvidia GPGPUs and they report speedups of up to 2.3x compared to state of the art quad core processors. Two of the three dimensions are decomposed using a variation of the same six step algorithm described in [3]. The intermediate data structure is a 5D array 256x(16x16)x(16x16) that maps naturally to machine parameters (16=half warp). This array is transposed 4 times using efficient memory accesses that can be coalesced for minimum latency. Even though there are other approaches with fewer memory accesses, the one chosen by the authors maps better on the hardware and improves overall performance.

Through the use of the six step algorithm the authors obtain shorter transforms with data fitting into the registers. Thus, there is no need to store intermediate results in memory (improved locality). Also, the execution of many small FFT transforms provides the opportunity to have the very large number of independent threads that a GPGPGU requires to execute efficiently (improved parallelism). From the data available it is hard to estimate the precise number of memory accesses that this implementation performs. The lower bound on the memory accesses is the number of elements of the FFT transform that is $O(Z^2)$ (in order to be able to compare to the other two architectures, in our evaluation we assume as if the authors would compute a single FFT transform of size $N = N_1 \times N_2 = Z \times Z$).

When comparing in terms of overhead instructions, we have to look at address calculation and loop control instructions. One complete FFT transform is performed without any loop so basically, it is as if the inner loop in Figure 3 didn't have any loop control instructions. There is no implicit address calculation, addresses are based on thread ID but to obtain the complete address, actual instructions must be executed. However, since all the data for the inner loop is inside registers, there is no need for address generation within the inner most loop. Another thing to note is that registers cannot be

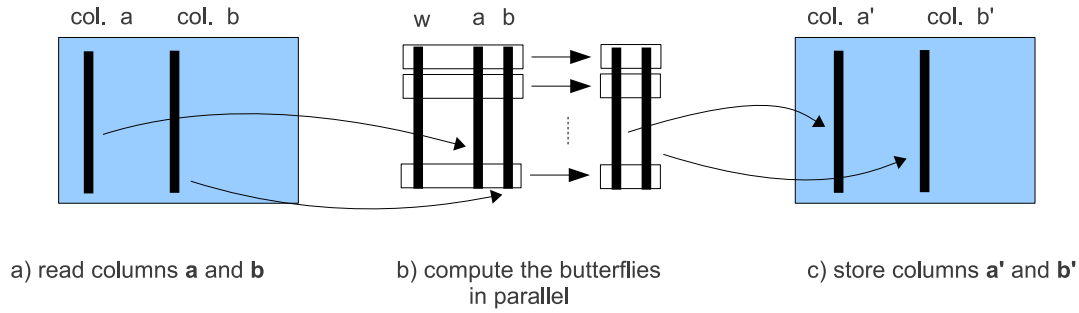

Fig. 5. The inner loop of a general vector processor implementation of the six step algorithm. Columns of the matrix are loaded in vector registers. The i^{th} butterfly is computed using the elements of vector registers on position i.

TABLE I
COMPARING THE THREE ANALYZED PLATFORMS IN TERMS OF VARIOUS FFT METRICS.

	memory accesses	dynamic instruction count	overhead instructions
Vector processors	$O(Z^2 \times log(Z))$	$O(Z \times log(Z))$	$O(Z \times log(Z))$
SIMD machines	$O(Z^2 \times log(Z))$	$O(Z^2 \times log(Z))$	$O(Z^2 \times log(Z))$
GPGPUs	$O(Z^2)$	$O(Z \times log(Z))$	$O(Z \times log(Z))$

accessed indirectly and thus register indexes are encoded in the assembly instructions and, as such, the number of dynamic instructions is equal to the number of static instructions. We estimate that the total number of overhead instructions is in the order of $O(Z^2)$. GPGPUs are large SIMD machines and every instruction executes a number of operations depending on the implementation (ie. NVIDIA Fermi executes 32 operations in one instruction) and thus the total number of executed instructions ranges between $O(Z^2 \times log(Z))$ and $O(Z \times log(Z))$. We shall use their best case of $O(Z \times log(Z))$.

Table I summarizes our estimations for the compared architectures (the amount of computation required by all architectures is the same, $O(Z^2 \times log(Z))$ since all of them implement the same underlying FFT algorithm of complexity $O(n \times log(n))$ with an input size of $n = Z \times Z$). The SIMD machines are the least efficient, they require memory instructions, overhead and total dynamic instructions of the same order as the computation. Vector processors have lower overhead and execute fewer instructions than the number of computations performed. Finally, the GPGPUs improve over the SIMD on the number of memory accesses and number of overhead instructions.

In the following section we will propose a simple architectural extension to vector processors that will enable an FFT implementation of Z independent transforms of length Z with memory accesses in the order of $O(Z \times Z)$, a number of overhead instructions in the order of $O(Z \times log(Z))$ and a number of total executed instructions in the order of $O(Z \times log(Z))$. Our proposal combines the benefits of all the studied architectures.

A method of generating the addressing sequence required by the FFT algorithms was presented in the US patent by Takano [13] but only applied to scalar general purpose processors. The proposed address generator produces a sequence of consecutive data locations. These consecutive addresses go through an address converter that rotates lower bits of the address according to a configuration register. Huang in [8] extends the SIMD register file with the possibility to perform element permutation with zero overhead. For SIMD register widths of up to 8, 32bit elements is assumed. To the best of our knowledge the permutation approach has not been previously applied to vector processors with long vectors support.

III. THE PROPOSED ARCHITECTURAL EXTENSION

In this section, we introduce the concept of a Permutation applied to a vector register and we shall then restate the Cooley-Tukey algorithm by using Permutations. An example for $Z = 8$ is shown and we shall discuss how to implement any transform size on a fixed machine vector length Z.

Consider Figure 3 and a vector processor implementation where the inner most loop executes step j of the i^{th} FFT transform.. If the block size of the six step algorithm is equal to the machine vector length Z, then after step j all the data needed for step $j+1$ are already in the vector registers but not in the required order.

We propose an extension to the state of vector registers that allows reading and writing them in various orders. For each vector register the programmer has access to a new configuration register called the Permutation Register, containing the Vector Permutation. The instruction set is extended with an instruction that writes the contents of the Permutation Register from one of the scalar registers. The instruction mnemonic is:

set_perm sreg_no, vreg_no

where $sreg\_no$ is the source scalar register and the $vreg\_no$ is the vector register to which the permutation is applied.

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: $(p_{k-1}, p_{k-2}, ..., p_1, p_0)$. 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';

We introduce two permutations that we will use to formulate the Cooley-Tukey algorithm in terms of permutations: the $shift(i)$ and $twiddle(i)$ permutations (Figure 6 a)). The $shift(i)$ permutation is defined as:

```
for j = 1 to k-1 do
  if j > i
    p_j = x_j
  else if j==i
    p_j = p_0
  else
    p_j = x_{j+1}
```

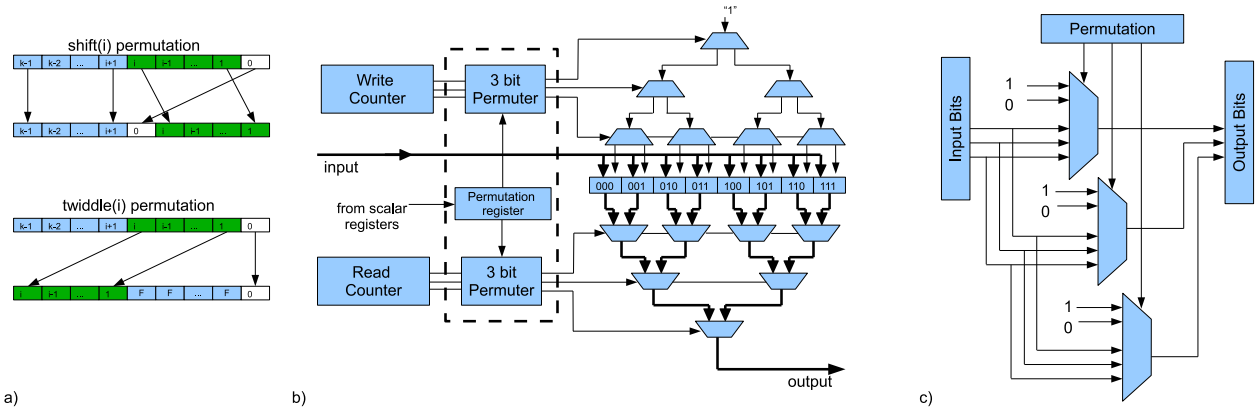

Fig. 6. a) The 'shift' and 'twiddle' permutations; b) The structure of a vector register extended with the permutation unit; c) The Permutation Unit is implemented using multiplexers.

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

Another permutation we will use is $twiddle(i)$ defined as:

```
threshold = k-1-i
p0 = x0
for j = 1 to k-1 do
  if j > threshold
     pj = xj−threshold
  else
     pj = 'F'
```

The Cooley-Tukey algorithm with Permutations. In the following paragraphs, we explain in detail how an FFT transform of length Z is executed on a vector processor with register permutation capabilities. The required $Z/2$ twiddle factors are precomputed and loaded in a vector register before the computation starts. All FFT transforms use the same twiddle factors and therefore during computation no memory bandwidth is wasted for twiddle factor transfers. Data for an FFT transform are loaded in vector registers before the computation begins and are stored back in memory after the entire transform has completed. All computations are performed on data in registers that do not spill into memory and thus, the implementation requires very low memory bandwidth.

The butterfly computation requires 3 complex inputs: the two data elements (a and b) and the twiddle factor w. It produces 2 complex outputs $a + (b * w)$ and $a - (b * w)$. We propose grouping the 3 pairs of inputs into 3 vector registers as follows (Figure 7): One register containing the real part of the data array, a second register containing the imaginary part of the data array while the third register contains alternating real and imaginary parts of the twiddle factors. The number of twiddle factors is equal to the butterfly computations required which in our case is equal to $Z/2$.

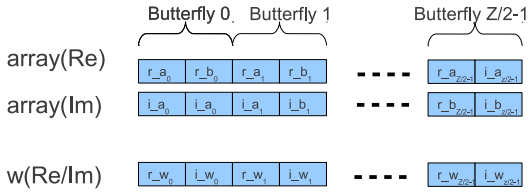

Fig. 7. Storing the data array and twiddle factors in three vector registers.

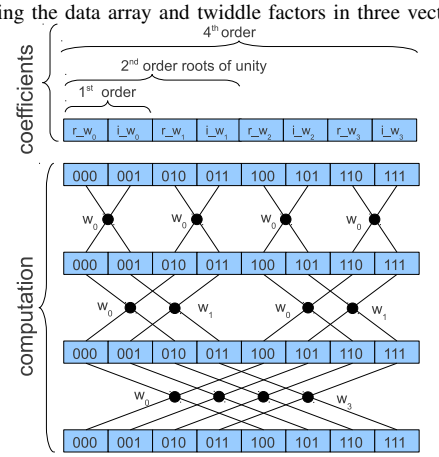

Fig. 8. The twiddle factors and the butterfly operations required by the 8 point FFT using the Cooley-Tukey algorithm.

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.

We can now express the Cooley-Tukey FFT algorithm in terms of the Permutations we introduced in the previous paragraphs that are applied to the input array and to the twiddle factors (see Figure 9). The bit reversal phase consists of $log(Z)$ consecutive permutations. Since mathematically, any number of consecutive permutations can be compressed into a single permutation, then all these permutations can be folded into the first data permutation for the computation phase. Through the

```
load input data and twiddle factors
//bit reversal
for i = 0 to log(Z) do
  set_perm shift(i) to data vector registers;

//computation
for i = 0 to log(Z) do
  set_perm shift(i) to data vector registers;
  set_perm twiddle(i) to coefficients vector register;
  compute the butterflies;
store results
```

Fig. 9. The Cooley-Tukey algorithm expressed in terms of permutations.

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

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

use of vector permutations, we obtain the bit-reversed input at zero cycles cost.

During the computation phase, only the permutation applied to the vector registers changes. The simple permutation logic ensures that for all the $log(Z)$ steps the $Z/2$ butterflies arrive at the functional units in the correct order and with the correct twiddle factors.

Let us have a close look at an example of an 8-point FFT transform, illustrated in Figure 8 and the permutations and address sequence they produce as shown in Table II. For the data array, the butterflies are computed in the 1^{st} step between elements at consecutive locations, for the 2^{nd} step elements that are two positions apart and for the 3^{rd} step elements that are 4 positions apart and so on. This is why the data permutation of step i, the i^{th} LSB alternates first. About the twiddle factors, the 1^{st} step computes 4 2point FFT transforms and requires a single twiddle factor: the 1st order root of unity that is replicated for all 4 butterflies. The 2^{nd} step computes 2 4point transforms that require two twiddle factors: the 2^{nd} order roots of unity. The 3^{rd} step computes a single 8 point transform that requires 4 twiddle factors: the 4^{th} order roots of unity. Since the set of 4^{th} order roots of unity includes the sets of 2^{nd} and 1^{st} order roots of unity, the problem of assigning the correct twiddle factor at the correct butterfly and at the correct step is a matter of index generation. This is handled by the $twiddle(i)$ permutation. Assuming that we have the 4^{th} order roots of unity in a vector register (Figure 7,8) then by applying permutation $twiddle(0)$ at the output we obtain only the 1^{st} order roots of unity replicated 4 times, for permutation $twiddle(1)$ we obtain the 2^{nd} order roots of unity replicated twice and for permutation $twiddle(3)$ we obtain the 4^{th} order roots of unity appearing only once each.

This approach can be extended for any machine dependent $Z = 2^k$. Any transform size N can be segmented using the six step algorithm into $N = Z \times N_1$.

Micro-architectural implementation. Figure 6 b) depicts a possible implementation of a vector register that is mirrored for reading or writing the register. When the register starts being read, the read counter is reset to zero and each cycle it is incremented until it reaches the value $Z - 1$. The value of the counter is used as selection bits for a MUX with Z inputs. Therefore, in Z consecutive cycles, the output of the MUX will correspond to all the elements of the vector register. A similar mechanism can be used for writing the vector register, we have chosen to show the DMUX implementation for symmetry reasons.

The micro-architectural extension we propose (marked by the dashed square in Figure 6 b)) consists of a Permutation unit that is added in between the counter and the selection bits of the MUX. Every bit in the output can be either of the input bits or 'T' or 'F'. Figure 6 c) presents a possible implementation of the Permutation Unit. The Permutation Register is set by software by copying the contents of a scalar register using the **set_perm** instruction.

IV. DESIGN EVALUATION

In this section, we compare the Cooley-Tukey FFT implementation on a vector processor with permutation support against a vector processor without. We start our evaluation by analytically determining the amount of cycles required to compute the FFT transforms in the two cases discussed so far. For simplicity, we only look at the combine phase of the algorithm, namely, the computation of the butterflies.

We assume the following instruction execution time (ml stands for memory latency and bl stands for butterfly latency):

- compute and set permutation word: 2 cycles
- address calculation: 2 cycles
- load data into a vector register: $(ml + Z)$ cycles
- load a single data element: ml cycles
- store a vector register: Z cycles
- butterfly operation: $(bl + Z)$ cycles

We use conservative timing for load operations, we assume that a load from linear addresses takes an equal amount of cycles as a strided load operation. We also assume the ability of the IBM 370 to perform two memory operations at once.

Based on the algorithm presented in Figure 4, we write the assembly level pseudo code for the FFT transform on the baseline vector architecture (in our case the results will be stored in the source operands, therefore a=a' and b=b' from Figure 3). This is depicted in Figure 10.

For the Temperton implementation [15] we assume that within the inner loop all data transfers are hidden behind the data computation. Thus, the lower bound on the amount of cycles required to perform the inner loop (both the data transfers and computation) is $2 \times Z$ cycles, the time required to transfer four vector registers using two memory units. We also assume that through careful programming, all memory latency can be hidden behind computation. The inner loop is executed Z times and loads the arrays, computes the butterfles

```
for j = 0 to log(Z) do
  for k = 0 to Z/2 do
    calculate addr of a = f1(j,k)
    calculate addr of b = f2(j,k)
    calculate addr of w = f3(j,k)
    load Re(w)  (one single value)
    load Im(w)  (one single value)
    for i = 0 to Z do
      load Re(a) (an array of Z values)
      load Re(b) (an array of Z values)
      load Im(a) (an array of Z values)
      load Im(b) (an array of Z values)
      (a, b) = butterfly(a, b, w)
      store Re(a)
      store Im(a)
      store Re(b)
      store Im(b)
    end
  end
end
```

Fig. 10. Assembly pseudo-code for computing Z independent transforms of size Z on the baseline vector machine.

```
load twiddle factors in vector register w
for i = 0 to Z do
  load Re(line i) into vector reg a
  load Im(line i) into vector reg b
  for j = 0 to log(Z) do
    calc. and set the data perm. for step k
    calc. and set the twiddle factor perm.
    for k = 0 to Z do
      (a, b) = butterfly(a, b, w)
    end
  end
  store Re(line i) from reg a
  store Im(line i) from reg b
end
```

Fig. 11. Assembly pseudo-code for computing Z independent transforms of size Z on the vector machine with permutation support.

and stores the results. The middle loop in additoin calculates addresses, loads twiddle factors and is executed $Z/2$ times. The total number of cycles is thus (inner loop within round brackets, middle loop within square brackets):

$$TC_{baseline} = [(2 \times Z) + 3 + 1] \times Z/2 \times log(Z)[cycles] \quad (1)$$

The number of memory accesses is:

$$TM_{baseline} = (8 \times Z + 2) \times (Z/2) \times log(Z)[memory\ acceses] \quad (2)$$

The number of address calculation instructions is:

$$TAC_{baseline} = 3 \times log(Z) \times Z/2[instructions] \quad (3)$$

The number of loop iterations is:

$$TL_{baseline} = Z \times Z/2 \times log(Z)[iterations] \quad (4)$$

In Figure 11, we adapt the algorithm from Figure 3 to use our architectural extension. As for the estimation for the baseline machine, we assume that careful programming can hide all memory transfers behind computation. Please note that this assumption does not put our architectural extension at an advantage over the baseline. The inner loop is executed Z times and computes the butterfles. In addition, the middle loop calculates and sets the permutations and is executed $log(Z)$ times. The outer loop adds the loading and storing of the data array and is executed Z times. The total number of cycles is thus:

$$TC_{perm} = \{[(Z + bl) + 4] \times log(Z)\} \times Z + Z[cycles] \quad (5)$$

The number of memory accesses becomes:

$$TM_{perm} = Z + Z \times Z \times 4[memory\ acceses] \quad (6)$$

The number of address calculation instructions is (the computation of a permutation is comparable to address calculation):

$$TAC_{perm} = 2 \times log(Z) \times Z[instrctions] \quad (7)$$

The number of inner loop iterations (each loop execution requires compare and jump overhead instructions):

$$TL_{perm} = Z \times Z \times log(Z)[iterations] \quad (8)$$

An important difference between the two approaches is that for the permutation implementation, each vector register holds elements for $Z/2$ butterflies while in the baseline case, each vector register holds data for Z butterflies. So, compared to the permutation, the baseline executes in the inner iteration twice as many butterflies but executes the outer iterations half as many times (Equation 1 and Equation 5). Because of this constant, the permutation is slightly slower than the baseline vector but bringing the benefit of significantly reduced number of memory accesses.

Since contemporary designs are more constrained by the memory bandwidth rather than by the available computational resources, we argue that for a given bandwidth we can potentially improve performance by allocating more hardware resources to utilize the saved bandwidth. Overall, we allow the possibility to trade-off between available memory bandwidth and hardware resources. By keeping the bandwidth constant and by using more hardware resources, we can potentially improve the FFT performance of vector processors by up to a factor of $log(Z)$. Since every transform is independent from the others, the amount of parallelism is very high and there are ample opportunities for performance improvements. We can reduce execution time by adding more functional units on a vector processor or by using more vector processors sharing the same memory bandwidth or a combination of both. We plan to investigate these possibilities in our future work.

The top line on Figure 12 plots the relative reduction of memory bandwidth of the permutation implementation compared to the baseline; the lower line plots the maximum speedup that can be gained for a constant bandwidth budget. We obtain these numbers by multiplying the relative bandwidth reduction (Equations 2 and 6) with the speedup (Equations 1 and 5). We consider typical values of section size varying from 16 to 256 elements and a butterfly unit latency of 20 cycles. For a small $Z = 16$, by allocating sufficient hardware resources to saturate the available bandwidth, our approach can be potentially up to 2.1x faster than a baseline implementation using the same bandwidth. For $Z = 256$, we can be up to 7.5 times faster than the baseline for the same memory bandwidth budget. Note that our approach is more

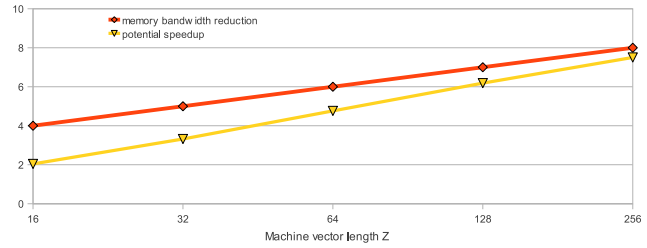

Fig. 12. Relative bandwidth reduction of the vector register permutation versus the baseline (top); and maximum potential speedup that can be obtained for a constant memory bandwidth budget (bottom).

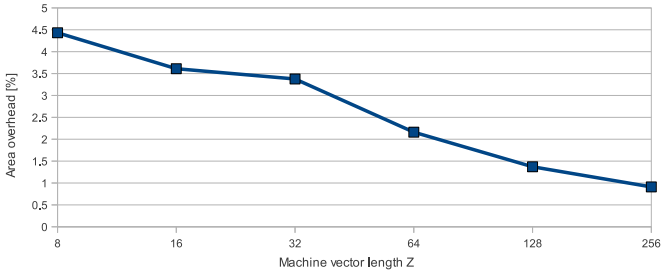

Fig. 13. Register file area overhead of the permutation logic, in percentages.

beneficial when vector lengths increase because the bandwidth savings are higher and because the difference in execution time reduces.

Synthesis results. We have implemented in Verilog a vector register file with and without the permutation support. We have used the Cadence RTL Compiler and we have synthesized for a 90 nanometer technology. The critical path for the register file is determined by the counter, the permutation unit and the multiplexers (Figure 6 b)). The permutation unit adds around $2 \times log(log(Z))$ levels of logic gates corresponding to a multiplexer with log(Z) inputs (see Figure 6 c)).

The processor's cycle time is determined by the delay of a 32bit adder in the scalar unit. The critical path of the register file including the permutation unit fits within the processor's cycle time for all the considered values of Z (8 to 256). Therefore, the permutation unit does not influence the cycle time.

Figure 13 plots the register file area overhead due to the permutation logic for various values of Z. The area overhead is for each vector register in the order of $log(log(Z))$ multiplexers each with $log(log(Z) + 2)$ inputs. The size of the vector registers increases much faster than the size of the permutation logic and, therefore, as Z increases, the area overhead decreases. The irregularity in the graph for $Z = 32$ is caused by increased logic depth that forced the tool to choose larger size components for fulfilling the timing requirement.

V. Conclusion

We proposed an architectural extension to the IBM System/370 Vector Architecture that enabled access to vector register elements in various orders under software control. We introduced the concept of permutation applied to a vector register and evaluated its flexibility and the related overhead. We investigated the Cooley-Tukey FFT algorithm execution of Z independent transforms of length equal to the machine vector length. We showed that the bit-reversal of the input can be folded into the computation at zero cost and how this FFT algorithm can be vectorized in a straight forward fashion. We also showed that our proposal reduced the required memory bandwidth by a factor of $log(Z)$ compared to the baseline implementation due to better register re-use. Under constant bandwidth and by using more hardware resources we can potentially improve the vector processors FFT performance. The additional cost of the permutation logic is trivial and does not penalize the overall vector cycle time. At the algorithmic level, the execution time has equivalent asymptotic behavior. Our results suggest that for small input sizes the execution time is longer than for traditional vector processors, however, as the input size increases, the difference in execution time becomes smaller while obtaining a considerable reduction of memory bandwidth.

References

[1] T. Endo-S. Matsuoka A. Nukada, Y. Ogata. 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] R.M. Smith-W. Buchholz A. Padegs, B.B. Moore. The IBM System/370 Vector Architecture: Design Considerations. In *IEEE Transactions on Computers 37*, pages 509 – 519, 1988.

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

[4] L.K. John D. Talla. Cost-effective hardware acceleration of multimedia applications. In *International Conference on Computer Design, ICCD*, pages 415 – 424, 2001.

[5] A.C. Arpaci-Dusseau J.L. Hennessy, D.A. Patterson. *Computer architecture: a quantitative approach*. 2007.

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

[7] I. Kuroda K. Nadehara, T. Miyazaki. Radix-4 FFT implementation using SIMD multimedia instructions. In *Acoustics, Speech, and Signal Processing, ICASSP*, pages 2131 – 2134, 1999.

[8] Z. Wang-W. Shi N. Xiao S. Ma L. Huang, L. Shen. SIF: Overcoming the limitations of SIMD devices via implicit permutation. In *High Performance Computer Architecture (HPCA) 16*, pages 1 – 12, 2010.

[9] N.P. Jouppi P. Ranganathan, S. Adve. Performance of image and video processing with general-purpose processors and media ISA extensions. In *26th International Symposium on Computer Architecture*, pages 124–135, 1999.

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

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

[12] 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.

[13] Hideto Takano. Hardware arrangement for fast Fourier transform having improved addressing techniques. In *US patent 5,430,667*, 1993.

[14] 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.

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

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