COMPUTATIONAL SIMD FRAMEWORK: SPLIT-RADIX SIMD-FFT ALGORITHM, DERIVATION, IMPLEMENTATION AND PERFORMANCE

Paul Rodríguez V. [1], *Marios S. Pattichis*[1] *and Ramiro Jordan*[2]

[1] ivPCL Laboratory & [2]ISTEC, University of New Mexico, Albuquerque, NM87131, USA
[prodrig,pattichis,rjordan]@eece.unm.edu

Abstract: A general framework to develop efficient Single Instruction Multiple Data (SIMD) compliant algorithms was recently proposed [3]. In this paper a split-radix SIMD-FFT algorithm is derived under this framework and compared against the lately developed radix-2 SIMD-FFT [1,2] algorithm, proven to have very efficient implementation. Regardless of the intrinsic irregular pattern present in the split-radix algorithm, it is shown that its performance improvement, when compared to the radix-2 algorithm, ranges from 2.5% upto 8.1%

1. INTRODUCTION

A SIMD capable processor can operate over S input elements in parallel, given that they are continuous in memory. The number S depends on the data type and on the target architecture; for single-precision floating-point numbers S is four (currently state-of-art for Intel's SSE & SSE2 [10] and Motorla's AltiVec[11]); without loss of generality, this is the considered data type.

A recent research on algorithms to compute the FFT of N-D complex input data [1,2], based on SIMD operations and on the classic radix-2 FFT algorithm, lead to a general framework to develop SIMD compliant algorithms [3]. The previous developed algorithms were found to be faster than any scalar FFT implementation as well other FFT implementations that take advantage of the SIMD architecture [6,7,8,9]

In this paper a novel split-radix SIMD-FFT is derived and implemented under the computational framework for SIMD architectures [3], and compared against the radix-2 SIMD-FFT; for N (complex input data size) equal or greater than 1024 elements the split-radix shows an improvement raging from 2.5% up to 8.1%. It should be recalled that the computational complexity, measured in flops, for the scalar radix-2 and split-radix algorithms are $5*N*\log_2(N)$ and $4*N*\log_2(N)$ respectively; so a 25% improvement (best case) should be expected, if memory access patterns are neglected.

This paper is organized as follows: in section 2 a briefly overview of the computational framework for SIMD architectures is presented, along with the programming model and the idea of the simd-flop cost; The radix-2 and split-radix SIMD algorithms are derived in section 3, where implementations issues are also discussed. In section 4 the computational results are shown; finally conclusions are listed in section 5.

2. COMPUTATIONAL SIMD FRAMEWORK

Overall SIMD technology has similar capabilities among different microprocessor manufacturers, nevertheless the manner a particular SIMD operation is carry out depends deeply on the particular architecture. If it is desired to develop a SIMD aware algorithm that will be easily implemented on any architecture it is necessary to understand the common functionality and constraints imposed on simd-operations. This understanding is also strongly related to the method used to code a program which employs simd-operations: it can depend on the compiler (i.e.: Intel's compiler, Motorola's patched GCC) and access simd-operations at a high level (i.e: simd_add, simd_mul, etc.) or it is architecture-dependent and accesses simd-operations at low level using a standard compiler (mixing assembly with the chosen programming language). The second approach is used in this framework, because an abstraction layer and primitive-simd-operations are easily developed, to map a particular operation into a set of architecture-dependent simd-operations.

Any SIMD capable processor has a set of special registers, whose characteristics (length and number) are architecture dependent (8 and 32 simd-registers, 128-bit long each for Intel's SSE and Motorola's AltiVec respectively). Note that with the chosen framework an efficient use of the simd-registers can be premeditated and may as well minimize the number of memory data access.

Memory data access has a great impact on the performance of any SIMD application (load a simd-register with memory data and vice-versa): addressed memory should be 16-bit aligned and data elements also should be continuous in memory; moreover the memory addressing mode depends on the architecture: AltiVec uses memory indexing mode for any memory address offset, while SSE specifies an immediate operand to accomplish the same task. Thus all pre-computed data (twiddle factors in the FFT algorithm for instance) should be arranged taken this constraint into account.

As an example of these facts, Table 1 shows how to perform a complex addition and subtraction of two complex operands and also multiplication of two complex operands (both are basic operations carried out in any FFT algorithm) for SSE and AltiVec architectures; also this table serves to the purpose of introducing the differences between both architectures, and how a primitive operation can be mapped for each architecture.

A simd-flop is defined as a floating-point operation carried out using the SIMD architecture; a simplistic relationship between a simd-flop and a (scalar) flop is

0-7803-7503-3/02/$17.00 ©2002 IEEE

that a simd-flop is equivalent to S scalar flops, where S is the number of elements that can be operated at the same time (S is equal to four for single-precision floating-point numbers); this relationship is verified if we calculate the number of flops needed for the scalar and simd case (for the Intel architecture we have 16 flops and 24 flops for addition/subtraction and multiplication respectively in the scalar case, and 4 and 6 simd-flops using SSE). Nevertheless, it is clear from Table 1 that depending on how a particular operation is mapped into the SIMD architecture it can use more or less memory access and floating-point operations as well; under the proposed framework, the number of memory accesses (kept to minimum) is trade-off by using more simd-registers.

Table 1

Input	X = A + jB	A,B,C,D are vectors of four
	Y= C + jD	contiguous elements in memory
Output	$O_1 = X + Y$	(element wise) $O_3 = X * Y$ (element wise)
	$O_2 = X - Y$	(element wise)

SSE	
Complex add/sub	Complex mul
load reg0 ← A	load reg0 ● A
load reg1 ● B	load reg1 ● B
load reg3 ← C	load reg3 ← C
load reg4 ● D	load reg4 ● D
load reg2 ● reg0	load reg2 ● reg0
load reg5 ● reg1	
	mul reg0 ← reg0 * reg3
add reg0 ● reg0 + reg3	mul reg2 ← reg2 * reg4
sub reg2 ● reg2 – reg3	mul reg3 ← reg3 * reg1
add reg1 ● reg1 + reg4	mul reg1 ← reg1 * reg4
sub reg5 ● reg5 – reg4	
	add reg0 ← reg0 – reg1
store RE(O_1) ● reg0	sub reg2 ← reg2 + reg3
store IM(O_1) ← reg1	
store RE(O_2) ← reg2	store RE(O_3) ← reg0
store IM(O_2) ← reg5	store IM(O_3) ← reg2

AltiVec	
load reg0 ← A	load reg0 ● A
load reg1 ← B	load reg1 ← B
load reg2 ● C	load reg2 ← C
load reg3 ← D	load reg3 ● D
add reg4 ← reg0 + reg2	set reg4 ← 0
sub reg5 ● reg0 – reg2	set reg5 ← 0
add reg6 ← reg1 + reg3	
sub reg7 ● reg1 – reg3	mac reg4 ← reg1 * reg3 + reg4
	mac reg4 ← reg0 * reg2 – reg4
store RE(O_1) ← reg4	mac reg5 ← reg0 * reg3 + reg5
store IM(O_1) ← reg6	mac reg5 ← reg1 * reg2 + reg5
store RE(O_2) ← reg5	
store IM(O_2) ← reg7	store RE(O_3) ● reg4
	store IM(O_3) ← reg6

In order to predict the time-performance of a given simd aware algorithm the time needed to perform a simd-flop could be used, nevertheless memory accesses have a great impact on the overall time performance, and must be taken into account. This issue is solved if a measurement of a complete basic operation, carried out repeatedly, in the given algorithm is elapsed; for the case of the SIMD-FFT the complex add/sub operation is chosen as the basic operation (see table 2, in section 4); this measurement not only give a simple way to predict the time performance, but also the relationship between the elapsed-time and the input data size, give hints on how to improve the particular implementation.

Also, it should be noted that any algorithm that accesses an array in a linear fashion (one element after the other) and performs similar operations over these elements can easily take advantage of any SIMD architecture. Thus existing algorithm should be modified to accomplish this basic constrain; the complexity of this task is algorithm dependent.

3. SIMD FFT

3.1 Radix-2 SIMD-FFT

The radix-2 SIMD-FFT algorithm modifies the operations performed in the first and second stage of the standard radix-2 FFT. Let $X = [x_0\ x_1\ ..\ x_{N-1}]^T$ where $N = 2^m$, then the radix-2 SIMD-FFT can be expressed as follows [1,2,5]:

$$Y = (\prod_{k=3}^{m} A_k) R_{22,N} T_{2,N} R_{21,N} S_N R_{12,N} R_{11,N} T_{1,N} S_N X \quad (1)$$

$$A_k = I_k \otimes B_{2^{m-k+1}} \quad (2)$$

$$B_{2L} = \begin{bmatrix} I_L & \Omega_L \\ I_L & -\Omega_L \end{bmatrix} \quad (3)$$

$$S_N = \begin{bmatrix} I_{N/2} & I_{N/2} \\ I_{N/2} & -I_{N/2} \end{bmatrix} \quad (4)$$

$$T_{1,N} = \begin{bmatrix} I_{3N/4} & 0 \\ 0 & -jI_{N/4} \end{bmatrix} \quad (5)$$

$$T_{2,N} = \begin{bmatrix} I_{N/4} & 0 & 0 & 0 \\ 0 & V_1 & 0 & 0 \\ 0 & 0 & I_{N/4} & 0 \\ 0 & 0 & 0 & V_2 \end{bmatrix} \quad (6)$$

Where $\otimes$ is the Kronecker product, I_N is an NxN identity matrix, $\Omega_L = \mathrm{diag}(1, W_{2L}, ..., W_{2L}^{L-1})$, $W_L = e^{-j2\pi/L}$ and $P_N = \mathrm{Per}(I_N)$ is the bit reversal permutation of the columns of the matrix I_N. $R_{11,N}=\mathrm{Mix}(I_2 \otimes P_{N/2})$ and $R_{12,N} = \mathrm{Mix}(I_2 \otimes \mathrm{Mix}(I_2 \otimes P_{N/4}))$; also $R_{21,N} = I_{N/4} \otimes P_4$ and $R_{22} = R_{11}$; The matrix operation Mix(H) is a permutation of the square NxN matrix H: let H be expressed as $H = [H_1, H_2, \ldots, H_N]^T$, where H_k is the k^{th} row of H, then $\mathrm{Mix}(H) = [H_1, H_{N/2}, H_2, H_{N/2+1}, \ldots, H_{N/2+1}, H_N]$.

Matrices V_1 and V_2 (equation (4)) are diagonal, where $V_1 = \mathrm{diag}(1, W_8^1, ..., 1, W_8^1)$. The elements of V_1 are composed of two factors, and each is repeated N/8 times. Also $V_2 = \mathrm{diag}(W_8^2, W_8^3, ..., W_8^2, W_8^3)$ has a similar structure. These matrices impose a restriction: the input data size must be greater or equal than eight.

It is well known that for the scalar case the complexity of the radix-2 FFT is $5*N*\log_2(N)$ flops; a similar analysis will confirm that the complexity for the SIMD (eq. 1) case is $5*(N/S)*\log_2(N)$ simd-flops.

3.2 Split-radix SIMD-FFT

Using a similar approach, the split-radix SIMD-FFT can be derived after analyzing the scalar split-radix algorithm, which ca be expressed as follows (partially based on [5, section 2.5]):

```
Y = P_N X;  loops = N/2;  L=1
for st=1:log2(N)
 for k=0:loops-1; L=L*2
  switch(β_N(k))
     case ONE:  Y(kL:(k+1)L-1) = S_L Y(kL:(k+1)L-1)
     case JAY:  Y(kL:(k+1)L-1) = J_L S_L Y(kL:(k+1)L-1)
     case _K:   Y(kL:(k+1)L-1) = K_L S_L Y(kL:(k+1)L-1)
     case _3K:  Y(kL:(k+1)L-1) = E_L S_L Y(kL:(k+1)L-1)
  end
 end
end                                                    (7)
```

Where J_L = diag(1,1,..1, -j, -j .. -j) has two groups of L/2 elements as shown. K_L = diag(W_{4L}^{S}) and E_L = diag(W_{4L}^{3S}), where s=(0:N/L:N-1) which are only needed for L = {2,4,..,N/4}; the array β_N contains the pattern needed to correctly perform the operations involved in the split-radix algorithm [5, section 2.5.5]

The split-radix SIMD-FFT algorithm modifies the operations performed in the first and second stage of the scalar split-radix FFT; for the first stage:

```
N4 = N/4; N34 = 3*N/4; N8 = N/8; N78 = 7*N/8;
X = S_N X                                              (8)
for k=0:4:N8-1
 switch( ψ_N/2(N8+k : N8+k+3) )
 case 1:  X(N34+k:N34+k+3) = K11 X(N34+k:N34+k+3)
 case 2:  X(N34+k:N34+k+3) = K12 X(N34+k:N34+k+3)
 case 3:  X(N34+k:N34+k+3) = K13 X(N34+k:N34+k+3)
 case 4:  X(N34+k:N34+k+3) = K14 X(N34+k:N34+k+3)
end                                                    (9)
X(6*N/8:7*N/8) = (-j)*X(6*N/8:7*N/8)                   (10)

for k=N4: 4 : N4+N8-1
switch( ψ_N/2(k:4) )
 case 1:  X(N78+k:N78+k+3) = E11 X(N78+k:N78+k+3)
 case 2:  X(N78+k:N78+k+3) = E12 X(N78+k:N78+k+3)
 case 3:  X(N78+k:N78+k+3) = E13 X(N78+k:N78+k+3)
 case 4:  X(N78+k:N78+k+3) = E14 X(N78+k:N78+k+3)
end                                                    (11)
X = R_12,N R_11,N X                                    (12)
```

Where $\psi_{N/2} = P_{N/2}\ \beta_{N/2}$, K_{11} = diag(1,1,1,1), K_{12} = diag(W_8, W_8, W_8, W_8), K_{13} = diag(1,1,W_8,1), K_{14} = diag(W_8, W_8, 1, W_8); also E_{11} = diag(-j, -j, -j, -j), E_{12} = diag($W_8^3, W_8^3, W_8^3, W_8^3$), E_{13} = diag(-j,-j,W_8^3,-j), and E_{14} = diag(W_8^3, W_8^3,-j,W_8^3)

For the second stage, let loops = N/2 and L=4, then:

```
X = R_22,N R_21,N S_N X                                (13)
for k=0:loops-1
 switch(β_N(k))
    case JAY:  X(kL:(k+1)L-1) = J_L X(kL:(k+1)L-1)
    case _K:   X(kL:(k+1)L-1) = K_L X(kL:(k+1)L-1)
    case _3K:  X(kL:(k+1)L-1) = E_L X(kL:(k+1)L-1)
 end
end
end                                                    (14)
```

The proposed split-radix SIMD-FFT algorithm will work for N≥32; nevertheless a similar approach is used for cases when N is 8 and 16. As in the radix-2 case, the minimum data size that can be processed using the ideas behind this algorithm is eight.

Operations defined for the first stage (eqs. 8-12) are easily mapped into primitive-simd-operations: all data accesses are linearly and contiguous in memory, S_N is the addition subtraction operation previously defined as well as the complex multiplication carried out in (9) and (11). It also should be noted that in the second stage, operations (13) and (14) can be merged into a single procedure, which is not shown here for the sake of simplicity.

Operations defined by $R_{11,N}$ and $R_{12,N}$ can also take advantage of the SIMD architecture, because they can be performed by accessing the high/low 64-bits of the SIMD register (that holds the partial result) and stored it in the appropriate memory location. From the implementation point of view, both operations ($R_{11,N}$ and $R_{12,N}$) are merged into a single operation. Also reordering the elements in the SIMD register can perform operations defined by $R_{21,\ N}$. The operation $R_{22,N}$ is the same as $R_{11,N}$.

For large N, the complexity of the scalar split-radix is $4*N*\log_2(N)$ flops [5]; also it could be proved that the complexity for the split-radix SIMD-FFT algorithm is $4*(N/S)*\log_2(N)$ simd-flops. From simd-flops only point of view, the performance improvement for the split-radix (compared to the radix-2) is 25%; nevertheless for both, the scalar and SIMD versions, the split-radix algorithm does not show a high regularity as in the radix-2 algorithm. Thus we can expect that its overall performance in a real application will be less than the theoretical one.

4. COMPUTATIONAL RESULTS

Both algorithms were implemented in C along with inline assembly instructions, using Linux as OS on an Intel architecture (to allow portability only PIII SSE instruction set was allowed) and on a Motorola PowerPC (PPC) architecture

The split-radix algorithm was fully tested on a PentiumIII (PIII-M with kernel 2.4.17; also the kernel was patched to be preemptible) running at 1.0 GHz, with 512M of RAM and 512K of L2-cache and on a Peintium4 (P4 with kernel 2.4.13) running at 1.4 GHz, with 512M of RAM and 256 of L2-cache. CPU clocks were measured using the time-stamp counter [10], and used to calculate the time performance of the split-radix and radix-2 SIMD-FFT implementations.

In table 2 the elapsed-time to complete a basic operation (complex add/sub) for both architectures is shown. It is important to note that the elapsed-time only follows a linear relationship with the input data size within ranges; if the relationship is broken ($N=2^{11}$ and $N=2^{14}$ for PIII and P4 respectively) then for the real implementation it is recommended to loop-unroll the

basic operation to improve the final time-performance (values in table 2 came from a loop version of the code shown in table 1). To predict the time performance of the SIMD-FFT algorithms the values shown in table 2 are multiply by (5/4)*$\log_2 N$ and $\log_2 N$ for the radix-2 and split-radix respectively (see italics in table 3).

Table 2

Time performance for the complex add/sub operation. The mean value for 10^4 iterations is shown.

SIZE	MEAN VALUE			
	PIII		P4	
	Ticks	Time (μs)	Ticks	Time (μs)
2^5	154.27	0.15	217.09	0.15
2^6	234.96	0.23	277.80	0.20
2^7	425.47	0.42	548.27	0.39
2^8	748.04	0.74	928.57	0.66
2^9	1407.25	1.40	1901.32	1.35
2^{10}	2886.14	2.87	3620.10	2.57
2^{11}	7490.05	7.44	7171.41	5.09
2^{12}	15138.13	15.04	14124.05	10.02
2^{13}	30169.38	29.97	28860.99	20.47
2^{14}	60735.20	60.33	243630.45	172.82

The procedure used to compare the time performance between both implementations was to perform the direct Fourier transform of complex-input data, for length from 2^5 up to 2^{14} elements for 1D arrays. The transform was performed repeatedly (10^4 iterations) for a particular size, and repeated 10 times. Also, any one-time initialization cost is not included in the measurements.

In Table 3 the time performance of the split-radix and the radix-2 SIMD-FFT are shown (best case). For small numbers ($N \leq 512$) the radix-2 version has a better performance (in the average); nevertheless for large numbers ($N \geq 1024$) that situation is reversed: the split-radix's improvement ranges from 2.5% unto 8.1%, where the percentage factor is: $100*(T_{RADIX\text{-}2}/T_{SPLIT\text{-}RADIX} - 1)$. This performance behavior can be explained recalling that the split-radix algorithm presents an irregular math operation pattern (compared to the radix-2 algorithm), and this fact affects its overall performance for mid-range numbers (128-512), whereas for large number, due to its smaller number of simd-flops (compared to the radix-2 case), its time-performance is improved. Also note that the estimated mean value for the time performance is accurate for the radix-2 case, whereas for the split-radix a significant difference is observed. The author expects, based on the estimated time-performance that the actual time-performance of the split-radix can be further improve.

5. CONCLUSIONS

The split-radix SIMD-FFT algorithm was derived and implemented following the computational framework for SIMD architectures introduced in [3]; its time performance shows an improvement over the radix-2 SIMD-FFT [1]; however, it is less than the theoretical bound predicted on a simd-flops based analysis; the author expects to get closer to the theoretical bound in a near future. Results shown in this paper, confirms that the memory access pattern and regularity of math operations, in any given algorithm, have a great impact in the overall time performance.

Table 3

Time performance for the 1D complex input data. The actual and predicted mean value (μs) for 10^4 iterations are shown.

SIZE	MEAN TIME (MICROSECONDS)			
	LINUX INTEL PIII (512M RAM 512K L2-CACHE)		LINUX INTEL P4 (512M RAM 256K L2-CACHE)	
	Split-Radix	Radix-2	Split-Radix	Radix-2
2^5	0.74 *(0.75)*	0.82 *(0.93)*	0.53 *(0.75)*	0.58 *(0.93)*
2^6	1.83 *(1.38)*	1.64 *(1.72)*	1.23 *(1.2)*	1.13 *(1.5)*
2^7	4.19 *(2.94)*	3.72 *(3.67)*	2.84 *(2.73)*	2.66 *(3.41)*
2^8	8.99 *(5.92)*	8.79 *(7.4)*	6.04 *(5.28)*	6.05 *(6.6)*
2^9	19.24 *(12.6)*	19.04 *(15.75)*	13.05 *(12.15)*	13.51 *(15.18)*
2^{10}	40.41 *(28.7)*	41.42 *(35.87)*	30.35 *(25.7)*	30.19 *(32.12)*
2^{11}	88.60 *(81.84)*	93.27 *(102.3)*	67.10 *(55.99)*	69.58 *(69.98)*
2^{12}	206.80 *(180.48)*	220.64 *(225.60)*	148.49 *(120.24)*	152.62 *(150.30)*
2^{13}	449.68 *(389.61)*	486.21 *(487.01)*	325.24 *(266.11)*	336.75 *(332.64)*
2^{14}	1010.32 *(844.62)*	1071.03 *(1055.8)*	855.34 *(560)*	884.44 *(844*)*

REFERENCES

[1] P. Rodríguez V. "A Radix-2 FFT Algorithm for Modern Single Instruction Multiple Data (SIMD) Architectures" accepted in ICASSP 2002

[2] P. Rodríguez V. "A Radix-2 Multidimensional Transposition-free FFT algorithm for Modern Single Instruction Multiple Data (SIMD Architectures" submitted to EUSIPCO 2002

[3] P. Rodríguez V., M.S. Pattichis, R. Jordan "Computational Framework for Single Instruction Multiple Data (SIMD) Architectures Applied to Digital Signal Processing" submitted to ICEE 2002

[4] D. E. Dudgeon, R. M. Mersereau "Multidimensional Digital Signal Processing" Prentice Hall, Englewood Cliffs, NJ 1984

[5] C. Van Loan "Computational Frameworks for the Fast Fourier Transform" SIAM 1992

[6] M. Frigo "A Fast Fourier Transform Compiler" Proceedings of the PLDI Conference, May 1999 Atlanta, USA

[7] F. Franchetti "Architecture Independent Short Vector FFT" ICASSP 2001 Proceedings, USA.

[8] "Split-Radix Fast Fourier Transform Using Streaming SIMD Extensions" Version 2.1 Application Notes Intel Ap-808 January 1999

[9] R. Crandall, J Klivintong "Super-Computing Style FFT Library for Apple G4" January 2000 Advanced Computation Group, Apple Computer

[10] "IA-32 Intel Architecture Software Developer's Manual" Vol. 2, No. 245471, 2001

[11] AltiVec Technology Programming Environment Manual – CT_ALTIVECPEM_R1 February 2001