A RADIX-2 FFT ALGORITHM FOR MODERN SINGLE INSTRUCTION MULTIPLE DATA (SIMD) ARCHITECTURES

Paul Rodríguez V.

Image and Video Processing and Communication Lab (ivPCL)
Department of Electrical and Computer Engineering
The University of New Mexico

ABSTRACT

Modern Single Instruction Multiple Data (SIMD) microprocessor architectures allow parallel floating point operations over four contiguous elements in memory. The radix-2 FFT algorithm is well suited for modern SIMD architectures after the second stage (decimation-in-time case). In this paper, a general radix-2 FFT algorithm is developed for the modern SIMD architectures. This algorithm (SIMD-FFT) is implemented on the Intel Pentium and Motorola PowerPC architecture for 1D and 2D. The results are compared against Intel's implementation of the split-radix FFT for the SIMD architecture [2] and the FFTW [3]. Overall, the SIMD-FFT was found to be faster than the other two implementations for complex 1D input data (ranging from 95.9% up to 372%), and for complex 2D input data (ranging from 68.8% up to 343%) as well.

1. INTRODUCTION

General-purpose processors with single instruction multiple data (SIMD) capabilities can process more than one data element in a single instruction. The SIMD architecture has rapidly become a standard feature in the past few years, and it is present in most general-purpose microprocessors (Intel, AMD, Motorola, etc.).

The well-known radix-2 FFT algorithm [7] is a very regular algorithm, from (i) memory access point of view, and (ii) complex math operations (performed over data in each stage) point of view. However, at the first stage of the radix-2 FFT (R2-FFT) algorithm, data must be accessed in the bit-reversal fashion (DIT case). This type of data-access breaks the continuity of the input data, and it will be difficult, using the SIMD architecture, to implement this algorithm in the standard way without losing performance, this happens because load/store and math operations are not done with SIMD operations.

A new general algorithm is proposed to compute FFT based on SIMD operations; the primary goals of this procedure are: (i) remove the bit-reversal data-access from the first stage, (ii) perform all operations involved in the first and second stage (of the standard R2-FFT algorithm) using SIMD operations, and (iii) let the procedure be recursive for handling lager size FFTs.

The target architectures were the Intel Pentium III processor and the Motorola 7400 PowerPC processor. The 2D version of the SIMD-FFT is implemented based on the 1D case (row-column approach) and on the Eklundh algorithm (matrix transposition) [7] that was also optimized for the SIMD architecture.

2. SIMD ARCHITECTURE

The support for SIMD instructions was introduced in general-purpose processors to improve the performance of different applications (multimedia, image processing, etc). The SIMD instructions were first introduced for integer data, and were extended, in the last three years, to support floating-point data. Any microprocessor with SIMD floating-point capabilities allows operations over four (current state of the art) single precision floating-point numbers (32-bit each) in a single instruction. In what follows, without loss of generality, only single precision floating point (32-bits) SIMD capabilities will be considered.

Floating point SIMD (FP-SIMD) capabilities have different names among different microprocessor manufacturers:

- Intel : "Streaming SIMD Extensions (SSE)" [1]
- AMD : "3DNow!" [4]
- Motorola: "AltiVec" [5]

Regardless of the manufacturer, processors with FP-SIMD capabilities, use a special set of registers (128-bit long each for SSE and AltiVec, 64-bit long registers for 3DNow!) to allow math operations over four single

0-7803-7402-9/02/$17.00 ©2002 IEEE

precision floating point numbers in a single instruction (this is true even for 3DNow! [4]).

3. RADIX-2 FFT

3.1. Classic algorithm

The Classic R2-FFT algorithm can be found in any textbook on signal processing [7]. In the present paper it is convenient to use the matrix framework introduced in [6]. Let $X = [X_0\ X_1\ ..\ X_{n-1}]^T$ where $n = 2^m$, then the radix-2 FFT of X can be expressed as [6, page 18]:

$$Y = FFT\{X\} = (\prod_{k=1}^{m} A_k)P_N X \quad (1)$$

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

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

where $\otimes$ is the Kronecker product [6, page 7], I_N is an NxN identity matrix, $\Omega_L = diag(1, W_{2L}, ..., W_{2L}^{L-1})$, $W_L = e^{-j2\pi/L}$ and $P_N = Per(I_N)$ is the bit reversal permutation of the columns of the matrix I_N. Note that the square matrix A_k represents the operations performed in the k^{th} stage of the radix-2 FFT algorithm.

3.2. SIMD approach: Radix-2 SIMD-FFT

The radix-2 SIMD-FFT algorithm modifies the operations performed in the first and second stage of the standard FFT; this can be expressed as follows:

$$Y = (\prod_{k=3}^{m} A_k)R_{22,N}T_{2,N}R_{21,N}S_N R_{12,N}R_{11,N}T_{1,N}S_N X \quad (4)$$

$$S_N = \begin{bmatrix} I_{N/2} & I_{N/2} \\ I_{N/2} & -I_{N/2} \end{bmatrix} \quad (5)$$

$$T_{1,N} = \begin{bmatrix} I_{3N/4} & 0 \\ 0 & -jI_{N/4} \end{bmatrix} \quad (6)$$

$$T_{2,N} = \begin{bmatrix} I_{N/4} & 0 & 0 & 0 \\ 0 & V_1 & 0 & 0 \\ 0 & 0 & I_{N/4} & 0 \\ 0 & 0 & 0 & V_2 \end{bmatrix} \quad (7)$$

where $R_{11,N} = Mix(I_2 \otimes P_{N/2})$ and $R_{12,N} = Mix(I_2 \otimes Mix(I_2 \otimes P_{N/4}))$; 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 $Mix(H) = [H_1, H_{N/2+1}, H_2, H_{N/2+2}, \ldots, H_{N/2}, H_N]$.

Matrices V_1 and V_2 (equation (7)) are diagonal, where $V_1 = diag(1, W_8^1, ..., 1, W_8^1)$. The elements of V_1 are composed of two factors, and each is repeated N/8 times. Also $V_2 = diag(W_8^2, W_8^3, ..., W_8^2, W_8^3)$ has a similar structure. These matrices impose a restriction: the input data size must be greater or equal than eight.

Any microprocessor with FP-SIMD capabilities can perform the operations defined by matrix S_N (equations (4) and (5)) using SIMD instructions (four additions/subtractions in a single instruction). Also the operations defined by $T_{1,N}$ and $T_{2,N}$ can be easily performed with SIMD instructions (four multiplications in one single instruction); furthermore, operations defined by $T_{1,N}$ does not need any multiplications and can be done by changing the sign of the data and storing the result in the adequate vector (real data goes to the imaginary part and vice versa).

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 that data in the appropriate memory location. From the implementation point of view, both operations ($R_{11,N}$ and $R_{12,N}$) are merged in a single operation. Operations defined by $R_{21,N}$ can be performed by reordering the elements in the SIMD register. The operation $R_{22,N}$ is the same as $R_{11,N}$

It must be noted that no assumption about the input data type (real or complex) was made; furthermore, operations define by $R_{22,N}$ $T_{2,N}$ $R_{21,N}$ S_N (second stage of the radix-2 SIMD-FFT) can be performed *in place*. Also, when the input data is real, operations defined in equation (4) can be re-arranged leading to a more efficient implementation (time and memory access point of view).

4. COMPUTATIONAL RESULTS

The algorithm presented in this paper was implemented in C along with inline assembly instructions, using Linux (kernel 2.4.6) as OS on an Intel architecture (to allow portability only PIII SSE instruction set was allowed) and on a Motorola PowerPC (PPC) architecture; its performance was compared against Intel split-radix FFT [2] (Intel architecture only) and the FFTW (version 2.1.3) [3] (in both architectures). While the radix-2 SIMD-FFT and the FFTW run under Linux, Intel code is targeted for Windows; hence, to make fair comparisons, the time-stamp counter (present in all Pentium processors), which is incremented every clock cycle [1], was used; the time-based register (MTB) [5] was used in the PowerPC case. Also the SIMD-FFT and the FFTW were compiled using

the Cygwin [8] utilities under Windows. The compilation options for the FFTW included the *–enable-i386-hacks* (Intel architecture only) and *–enable-float.* (both architectures). The Intel implementation of the split-radix is fixed (128 elements) and comes in three versions (source code point of view): assembly, Intrinsics and C++ vector classes.

The target machines were (i) P4 running at 1.4 GHz, with both OS: Linux and Windows 2000; its CPU clock was measured (both OS), using the time-stamp counter [1]; and (ii) 7400 PowerPC running at 450 MHz with Linux as OS; its CPU clock was measured using the time-based register. The procedure was (all cases) to count the increment of the time-stamp counter or time-based register in one second over 10^3 iterations.

The procedure used to compare the performance of all programs was to perform the direct Fourier transform of real-input data, as well as complex-input data, for length from 2^5 up to 2^{14} elements in the 1D case and for length from 2^5 up to 2^{10} elements in the 2D. The transforms were performed repeatedly (10^4 and 10^3 iterations for 1D and 2D respectively) for a particular size, and repeated 10 times. Also, any one-time initialization cost is not included in the measurements. Results (best case) are shown in tables I and II (also fig. 1 to 5) for the complex case, 1D and 2D respectively.

The Intel's code performance varies about 30% depending on the coding scheme (see figure 1). When comparing its performance against SIMD-FFT, the assembly case (which presented the best performance of the three Intel's implementations) is much slower than the implementation presented.

On Intel architecture the complex 1D SIMD-FFT's performance is better than the FFTW's performance ranging from 95.9% up to 374% (table I and figure 2); the percentage factor is: $100*(T_{FFTW}/T_{SIMD\text{-}FFT} - 1)$. These results outperform the results presented in [9], where the SIMD implementation of the FFTW outperforms the scalar FFTW implementation raging from 25% up to 50%. Also, for the complex 2D case, the SIMD-FFT outperforms the FFTW's scalar implementation ranging from 68.8% up to 343% (see table II and figure 4).

Also, on the PowerPC architecture the complex 1D SIMD-FFT's performance is better than the FFTW's performance ranging from 9.1% up to 118% (see table I and figure 3); these results are compatible with results presented in [10], nevertheless it must be pointed out that the AltiVec architecture on the 7450 PPC processor has been greatly improved when compared to the 7400 PPC processor. Also, for the complex 2D case, the SIMD-FFT outperforms the FFTW's scalar implementation rangingfrom 13% up to 92% (see table II and figure 5).

SIZE	1D FFT MEAN TIME (MICROSECONDS)			
	LINUX INTEL		LINUX POWERPC	
	SIMD FFT	FFTW	SIMD FFT	FFTW
2^5	0.58	1.14	1.08	1.96
2^6	1.13	2.26	2.24	4.17
2^7	2.66	5.59	5.06	10.76
2^8	6.05	12.52	11.08	24.19
2^9	13.51	29.46	24.39	51.96
2^{10}	30.19	68.62	56.86	118.92
2^{11}	69.58	170.18	134.28	275.26
2^{12}	152.62	603.71	390.08	609.36
2^{13}	336.75	1590.74	1242.28	1359.13
2^{14}	884.44	3374.00	2731.27	2981.54

Table 1. Time performance for the complex 1D input data. The mean value (μs) for 10^4 iterations is shown.

SIZE	2D FFT MEAN TIME (MILISECONDS)			
	LINUX INTEL		LINUX POWERPC	
	SIMD FFT	FFTW	SIMD FFT	FFTW
2^5	0.041	0.070	0.088	0.114
2^6	0.168	0.317	0.459	0.522
2^7	0.937	1.712	2.355	3.805
2^8	5.147	20.611	12.796	18.998
2^9	22.436	99.517	72.116	139.011
2^{10}	115.350	426.015	357.014	665.109

Table 2. Time performance for the complex 2D input data. The mean value (ms) for 10^3 iterations is shown.

It must be noted that SIMD-FFT's performance follows the same tendency in both Intel and PPC architecture (2^5 - 2^{10} data length); nevertheless for data lengths greater or equal than 2^{11} the performance improvement drops dramatically in the PPC case.

5. CONCLUSIONS

A new recursive general algorithm that computes the radix-2 FFT and takes advantage of the SIMD architecture was derived and implemented (1D and 2D); its time performance was found to be better than the performance of other implementations that make use of the SIMD architecture [2] as well as other more general implementations [3]. Results were tested over different OS, and different architectures.

6. ACKNOWLEDGEMENT

The author would like to thank Dr. Marios S. Pattichis, whose suggestions improved this work.

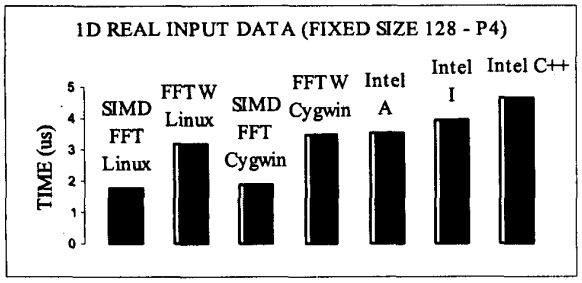

Figure 1. The time performance of the different FFT implementations (real input data) is shown for the special case of size 128 on the Intel architecture.

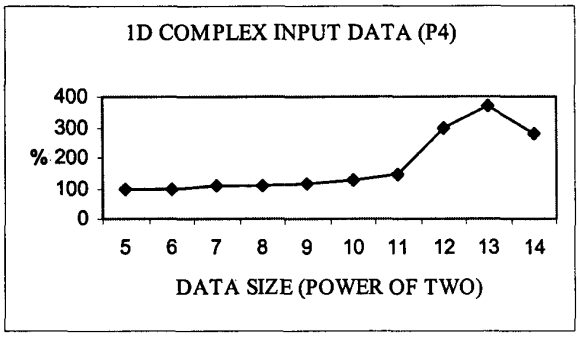

Figure 2. 1D SIMD-FFT performance improvement over the scalar 1D FFTW is shown for the Intel architecture.

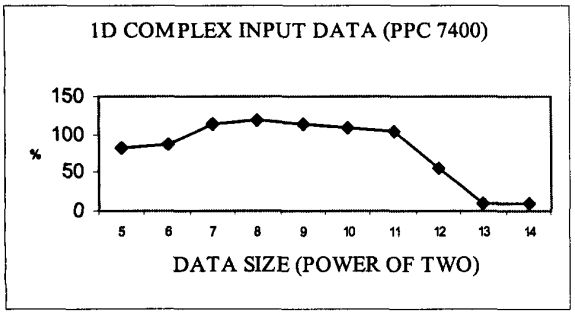

Figure 3. 1D SIMD-FFT performance improvement over the scalar 2D FFTW is shown for the PPC architecture.

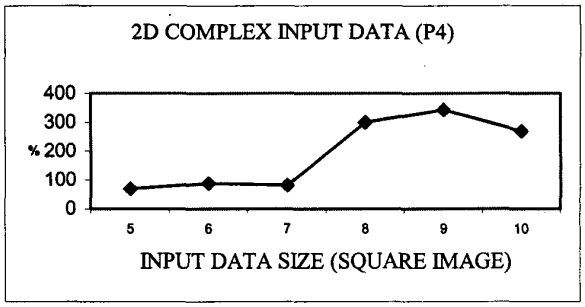

Figure 4 2D SIMD-FFT performance improvement over the scalar 2D FFTW is shown for the Intel architecture.

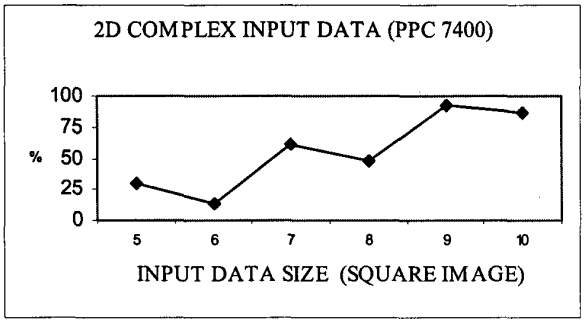

Figure 5 2D SIMD-FFT performance improvement over the scalar 2D FFTW is shown for the PPC architecture.

7. REFERENCES

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

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

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

[4] 3DNow Technology Manual - AMD No. 219280 March 2000

[5] AltiVec Technology Programming Environment Manual – CT_ALTIVECPEM_R1 February 2001.

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

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

[8] G. Noer "Cygwin: A free Win32 Porting Layer for UNIX applications" Proceedings of the 2nd USENIX Windows NT Symposium August 1998, Seattle, Washington, USA.

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

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