A FAST AND STALL-FREE MEMORY ACCESS FOR FFT COMPUTATIONS

Mokhtar A. Aboelaze *
Department of Computer Science
York University
4700 Keele st.
N.York, Ontario M3P 1J3 CANADA
tel: (416) 736-5053 ext 33947
E-mail aboelaze@cs.yorku.ca

ABSTRACT

Performing a 2-D FFT in real time require designing special purpose VLSI chips. One of the major problems in designing special purpose VLSI chips is to match the processor speed and the memory speed. Advanced pipelining and multiprocessing techniques have enabled us to shorten the processor cycle (pipeline cycle) beyond the limit, any memory chip can handle.

In this paper, we propose a method to allocate the N^2 points on an 2-D image to m memory modules in order to guarantee a stall-free access to the data in calculating 2-D FFT, even when the memory is m times slower than the processor. Our technique depends on accessing the memory modules in a round robin fashion, and generating the appropriate twiddle factor and memory address for every point accessed. Generation of the appropriate twiddle factors, and the memory addresses is simple and can be implemented in hardware using minimal cost.

Key Words: 2-D FFT, memory interleaving, conflict-free memory access, special purpose VLSI arrays.

* This work is partially supported by a grant from the National Science and Engineering Council of Canada number NSERC-OGP0043688

1. INTRODUCTION

Two-dimensional Fourier transform is a useful operation in many image and signal processing applications. Applications, such as radar signal processing, computer tomography, computer vision, and robotics require performing discrete Fourier transform in real time. General purpose computers are not suitable for this kind of applications, which lead to designing special purpose FFT chips to perform real time FFT [Eld88], and two-dimensional FFT [TaS88].

FFT of any sequence can be calculated by decomposing the sequence into m smaller sequences, recursively calculating the FFT of the smaller sequences, then combining the FFT of the smaller sequences to produce the FFT of the original sequence (combining the FFT of the smaller sequences is known as *butterfly* operation) [OpS75]. Two-dimensional FFT, as we will see in section 2, could be computed by computing a 1-D FFT for the rows of the image, followed by a 1-D FFT on the resulting columns, thus reducing 2-D FFT into a series of 1-D FFT operations.

FFT computations could be performed *in place*, where the inputs and the outputs of the butterfly operations have the same set of points; thus eliminating the intermediate storage requirements and reducing the amount of memory required by half [OpS75]. In cases where special purpose chips are used to calculate FFT, this is of a particular importance since the cost and speed of the chip depends on the amount of memory required on it. Reducing the memory requirements leads to a faster and less expensive chips.

In this paper, we envision performing 2-D radix m FFT using a special purpose hardware chip to implement the butterfly operations. We assume that the image is in the form of $N \times N$ matrix, where $N = m^r$, in this case, m is the radix, and r is the number of ranks. The 2-D FFT could be decomposed into 1-D FFT operations. Those 1-D FFT's are in turn decomposed into a length m butterfly operations. The objective of this paper is to perform each butterfly operation in the shortest possible time. The system consists of a processor chip, and m memory modules to store the image points. We assume that the processor could be implemented as a powerful pipelined adder/multiplier, where we supply it with new data item every pipeline cycle, or we can achieve a better performance by using m different pipelined adder/multiplier to perform the m operations in radix m FFT in parallel, in this case, we have to supply m data elements every pipelined cycle. In either cases, one memory module is usually not sufficient to match the processor pipeline cycle [HeP90] (we assume that the memory is m times slower than the processor). Therefore, we assume that the data are distributed into m different memory modules in order to improve the memory throughput.

Previous attempts for implementing fast FFT chips are mentioned in [CoS75], [TaS88], and [Cor71]. However, they did not address the problem of matching the processor speed and the memory speed. Johnson in [Joh92] addresses the problem of distributing the image points amongst the memory modules in order to have a conflict free memory access to the inputs of any butterfly operations. However, his solution did not eliminate the overlapping effect of performing two consecutive butterfly operations.

In this paper, we introduce two methods for distributing the image points amongst the memory modules in order to guarantee a conflict-free, steady stream of data to the FFT chip. In order to achieve this, we must be sure that not only all the inputs and outputs of any butterfly operation are allocated to different memory modules, but also we have to be sure that the overlapping among two butterfly operations will not cause any memory conflict. This is illustrated by the following example. Assume that $m = 4$, and assume that the memory modules storing the input to the first butterfly operation are M_0 , M_1 , M_2 , M_3, (in that order), and the memory modules storing the input to the second butterfly operation are M_3 , M_0 , M_1 , M_2 (in that order). From this example, it is clear that the input to the the two butterfly operations belong to different memory modules. However, the last memory module accessed in the first butterfly operation is M_3, while the first memory module accessed in the second butterfly operation is M_3, which results in stalling the pipeline since the memory access time is usually longer than the processor pipeline cycle time.

In this paper, we overcome this problem by forcing the memory modules access to be in the same order for every butterfly operation (M_0 , M_1 , M_2 , M_3). Since the inputs to the same butterfly operation are stored in different memory modules, then the problem is reduced to choosing the appropriate *twiddle* factor for every operation and to generate the appropriate address for every memory module. Generating the twiddle factor for every operation, and generating the address of the accessed element in every memory module could be implemented in hardware and with minimum cost.

The remaining of the paper is as follows. Section 2 is a quick review on 2-D FFT. Section 3 introduces two different methods for mapping the data into the different memory modules, one method is used when m is not a power of 2, the second method is used when m is a power of 2. Section 4 introduces the hardware required to generate the memory addresses and the appropriate twiddle factors (for the case of m is a power of 2). Section 5 is a conclusion.

2. 2-D FFT

One dimensional FFT of a sequence $x(i)$ $0 \le i < n$ is $X(i)$ $0 \le i < n$, where

$$X(i) = \sum_{j=0}^{n-1} x(j)\ \omega_n^{ij} \tag{1}$$

Where $\omega_n = e^{-j2\frac{\pi}{n}}$ is the n^{th} root of unity.

For a 2-D FFT, we calculate first 1-D FFT of the individual rows of the image, followed by 1-D FFT on the resulting image. Assume that the input image is $x(i_1, i_2)$ $0 \le i_1, i_2 < n$. To calculate 2-D FFT of the image, $X(i_1, i_2)$ $0 \le i_1, i_2 < n$, first, calculate the FFT of the rows

$$Y(i_1, i_2) = \sum_{j_2=0}^{n-1} x(i_1, j_2)\ \omega_n^{i_2 j_2} \qquad 0 \le i_1 < n \tag{2}$$

Then, the Fourier transform on the columns of the resulting image produces

$$X(i_1, i_2) = \sum_{j_1=0}^{n-1} Y(j_1, i_2)\ \omega_n^{i_1 j_1} \qquad 0 \le i_2 < n \tag{3}$$

As we have mentioned before, FFT can be recursively calculated by decomposing a sequence of N points into m sequences each with N/m points, calculating the FFT of the m sequences and combining the results to produce the FFT of the original sequence. Figure 1 shows the FFT of a 27-point sequence by recursively dividing the sequence to three groups. It also shows the memory module and the address in this memory module every point is stored in according to the method proposed in Section 3.

3. CONFLICT-FREE MEMORY ACCESS

In this paper, we assume that the memory access cycle is m times slower than the processor pipeline cycle (either one pipelined processor or m pipelined processors for the m complex multiplication in each butterfly operation). Our goal is to distribute the image points of the 2-D image amongst the m memory modules (m is the radix of the FFT) in order to guarantee a conflict free continuous stream of data to the processor(s) in order to perform the butterfly operations. We propose two different techniques, the first is used when m is not a power of 2, the second is used when m is a power of 2.

We assume that the image is on the form of a square matrix where $x_{i,j}$ is the element in the i^{th} row and j^{th} column. we assume that $z = i \,|\, j$; i.e. z is the number resulting from concatenating i and j. We also assume that $z_k \;\; 0 \le k < 2\log_m N$ are the base m digits of z. Notice that the input to any butterfly operation in the i^{th} level agree in all the digits except z_i for rows ($z_{i+logN-1}$ for columns).

3.1. Method I

In this method, we store element $x_{i,j}$ in location $\left\lfloor \frac{z}{m} \right\rfloor$ in memory module $M = \sum_{k=0}^{2\log_m N-1} z_k$ mod m. In the remaining of this subsection, we prove that using this mapping, no two elements that are used in the same butterfly operation are mapped to the same memory module, and no two elements that are mapped to the same memory module are mapped to the same memory location. Then we show how to calculate the twiddle factor and the memory address for every operation.

Lemma 1:

All the elements used in a butterfly operation are mapped to distinctive memory modules.

Proof: Each stage in the calculation of FFT contains $N/log_m N$ butterfly operations. In the $\emptyset^{th}$ stage of calculating FFT for a row of the image, the input to any butterfly operation z agree on all the base m digits of z except $z_\emptyset$ ($z_{\emptyset+\log_m N-1}$ for FFT of a column). Assume that 2 elements used in the calculation of a butterfly in the $\emptyset^{th}$ stage (x and y) are mapped to memory modules M_1 and M_2 respectively, such that

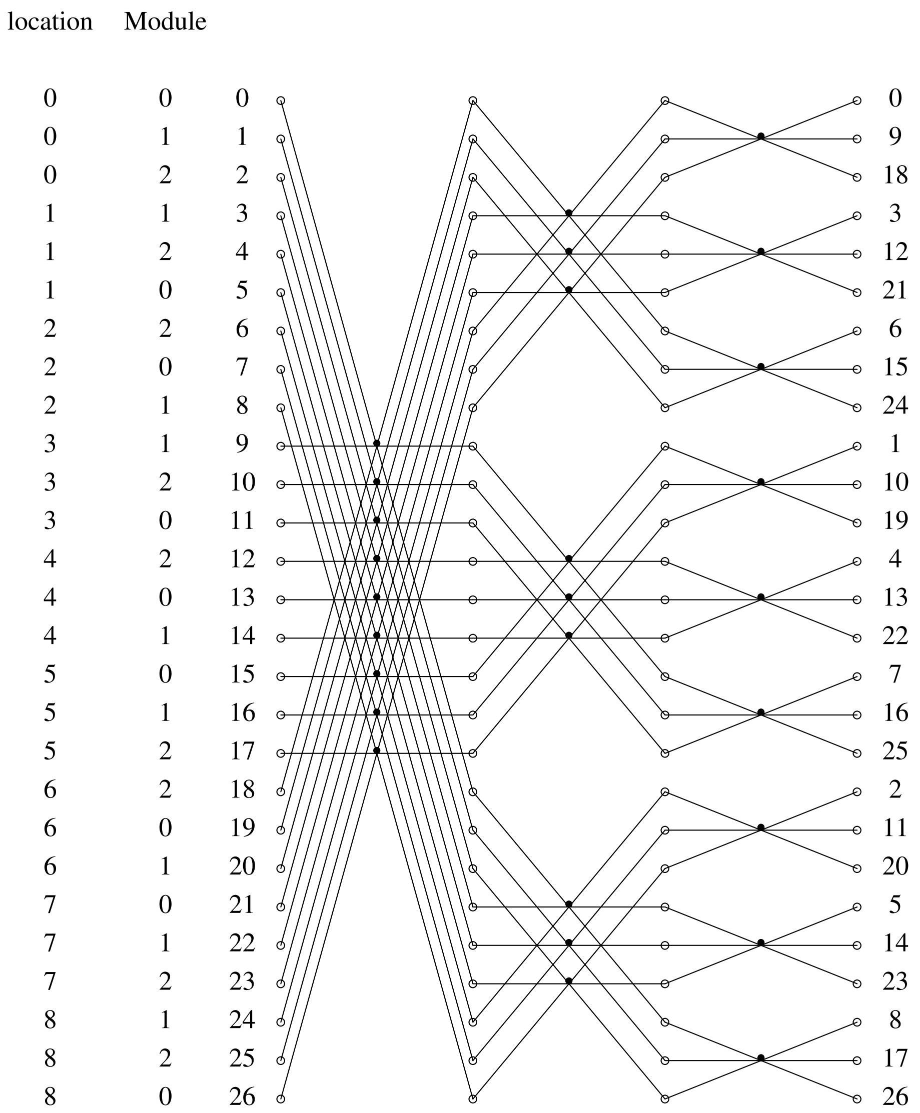

Figure 1
Computing 27 points FFT by recursively decomposing it into 3 point sequences.

$$M_1 = \sum_{\substack{k=0 \\ k \neq \emptyset}}^{2log_m N-1} x_k + x_\emptyset \quad M_2 = \sum_{\substack{k=0 \\ k \neq \emptyset}}^{2log_m N-1} y_k + y_\emptyset \tag{4}$$

Since x and y agree on all the base m digits except the $\emptyset$ digit, and since by definition $x_\emptyset \neq y_\emptyset$; then $M_1 \neq M_2$ $\square$

Lemma 2:

Any two points mapped to the same memory module are not mapped to the same memory address in this module.

Proof: Assume that two elements z_1, and z_2, are mapped to the same memory address *i.e.* $\lfloor z_1/m \rfloor = \lfloor z_2/m \rfloor$. The only possibility for this to happen is when both z_1 and z_2 agree in the $2log_m N - 1$ most significant digits of their base m representation, *i.e.* they are input to a butterfly in level 0 *i.e.* they are mapped to different memory modules. □

In each cycle, the processor scan the memory modules $M_0 M_1 \cdots M_{m-1}$ accessing the m inputs of a butterfly operation. The inputs of the butterfly operation in the i_{th} stage agree on all the digits except z_i. Since the memory modules are accessed in order every processor should be able to know which z_i it is getting, z_i for every processor could be calculated as follows: if the processor accesses memory module j, then, it is getting the pixel with z_i

$$z_i = \left\{ j - \sum_{\substack{k=0 \\ k \neq i}}^{2log_m N - 1} z_k \right\} \ mod\ m \tag{5}$$

Thus after accessing the element from each memory module, the appropriate twiddle factor is calculated and the butterfly operation is performed.

3.2. Method II

This method can be used when m is a power of 2, it is faster to generate the memory address and easier to implement. We store element $x_{i,j}$ in location $\lfloor z/m \rfloor$, where $z = i|j$. in memory module $m = a_0 \oplus a_1 \oplus \ldots \oplus a_{\log_m N-1} \oplus b_0 \oplus b_1 \ldots b_{\log_m N-1}$ where a_k and b_i are the binary representation of the base m digits of i and j respectively.

The proof that any two elements that are part of a butterfly operations are mapped to different memory modules, and that any two elements mapped to the same memory module are mapped to two different memory locations is similar to the proof in method I and will not be repeated here.

The memory location to be accessed in each memory module to get the elements required for a butterfly operation is $\left\lfloor \frac{z}{m} \right\rfloor$, for element $z = i\,|\,j$, while the twiddle factor in stage i for the element accessed from memory module number M is

$$i = \bigoplus_{\substack{k=0 \\ k \neq i}}^{m-1} a_k \bigoplus_{k=0}^{m-1} b_k \oplus M \tag{6}$$

Where the EX-OR is performed on the binary representation of a_k, b_k, and M respectively.

4. HARDWARE REQUIREMENTS

In this section we describe the hardware required to generate the addresses for the memory references and the twiddle factor when m is a power of 2. For the case of m is not a power of 2, the idea is the same, but we have to replace the EX-OR with adder/subtracter. For element x_{ij}, we assume that $a_0, a_1, a_{\log_m N-1}$ are the base m digits of i. Similarly $b_0, b_1, b_{\log_m N-1}$ are the base m digits of j. Also we assume that $z = i|j$. Consider accessing point z, we want to generate the following

(a) The address of the memory module we have to access.

(b) The address of the element in the chosen memory module.

(c) The appropriate twiddle factor.

Without any loss of generality, consider the $\emptyset^{th}$ stages in calculating a FFT of the rows. The input to the butterfly operations have the same binary representation of all the digits except $a_\emptyset$ ($b_\emptyset$ in case of columns). First, the memory modules are accessed in a round robin fashion $0\ 1\ 2\ \ldots\ m-1$, which could be generated by a simple modulo m counter. The address of the element in the memory module is simply $\lfloor z/m \rfloor$, which could be calculated in a binary form by ignoring the $\log m$ least significant bits of the binary representation of z. The appropriate twiddle factor could be generated by calculating Eq. 6.

Figure 2 shows how to calculate the appropriate twiddle factor by implementing Eq. 6. Equation 6 generates the twiddle factor by EX-ORing all the radix m digits of z except the $a_\emptyset$ digits with M the memory module number. This could be achieved by choosing $2m-1$ digits out of the $2m$ digits representing z using $2m-1$ multiplexors and $2m-1$ comparators. The input to each comparator is the stage number and the digit number. If the stage number is greater than the digit number, the multiplexes chooses the right input, otherwise, it chooses the left input (thus eliminating $a_\emptyset$).

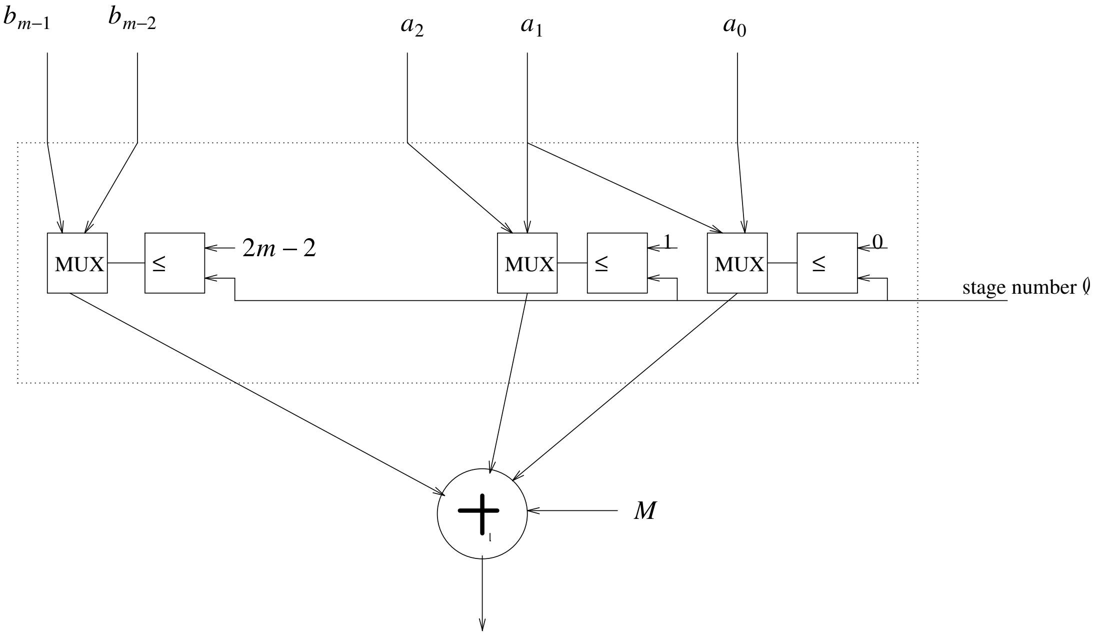

Figure 2
Generating the appropriate twiddle factor for the ℓ^{th} stage

It is worth mentioning here that since the input to the same butterfly agrees in all the digits except a_ℓ, which means the the output of the comparators and multiplexors is the same for all the elements required for the same butterfly, *i.e.* the output of the dotted box in Figure 2 can be calculated once for a butterfly then EX-ORed with M for $0 \le M < m$.

5. CONCLUSION

In this paper we introduced a method of distributing the points of a 2-D image in m memory modules in order to implement a radix m 2-D FFT. Our goal is to have a conflict-free, stall-free access to the image points in order to match the processor speed and the memory speed. Our method depends on accessing the memory modules in round robin fashion, and for every memory module access, we have to generate the appropriate address and the appropriate twiddle factor. generation of the address and the twiddle factor is simple and can be implemented in hardware with minimal cost.

REFERENCES

[CoS75] M. J. Corinthios, K. C. Smith, and J. I. Yen, ‘‘A Parallel radix-4 Fast Fourier Transform Computer,’’ *IEEE Transactions on Computers,* Vol. C-24, pp 80-92, January

1975.

[Cor71] M. J. Corinthios, "A Fast Fourier Transform for High Speed Signal Processing," *IEEE Transactions on Computers,* Vol. C-20, pp 843-846, August 1971.

[Eld88] J. A. Elden, "Fourier on a chip," in *VLSI Signal Processing, vol III,* R. W. Broderson and H. S. Moscovitz, eds., IEEE Press., 1988, pp. 15-20.

[HeP90] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach,* Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.

[Joh92] L. G. Johnson, "Conflict Free Memory Addressing for Dedicated FFT Hardware," *IEEE Transactions on Circuits and Systems,* Vol. 39, No. 5, May 1992, pp. 312-316.

[OpS75] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing,* Prentice-Hall, 1975.

[TaS88] G. F. taylor, R. H. Steinvorth, and J. F. McDonald, "An Architecture for a Video Rate Two-Dimensional Fast Fourier Transform Processor," *IEEE Transactions on Computers,* Vol. 37, No. 9, September 1988, pp. 1145-1148.