Using a Scalable Parallel 2D FFT for Image Enhancement

Yaniv Sapir Adapteva, Inc.
Email: yaniv@adapteva.com

Introduction

Frequency domain operations on spatial or time data are often used as a means for accelerating calculations. For example, when filtering a signal, one can use convolution to perform the operation in the spatial domain or transform the signal to frequency domain and apply a window function to achieve the same effect.

The problem with the first approach is that for large kernels, the convolution complexity approaches $O(N^2)$. When using a smart transform algorithm like the Radix-2 Fast Fourier Transform (FFT), the complexity is reduced to $O(N \cdot \lg_2 N)$. For a large N this can become a significant difference.

Another reason for performing an operation in the frequency domain is that designing the filter may be easier and more intuitive (as a very simplistic example, a simple low-pass filter is merely a nulling of the high order frequency components of the signal).

In this paper we will present the demonstration of a simple image enhancement program. This operation is performed in frequency domain by first applying 2D FFT to an image, then apply a low-pass filter and convert back to special domain by a 2D IFFT operation:

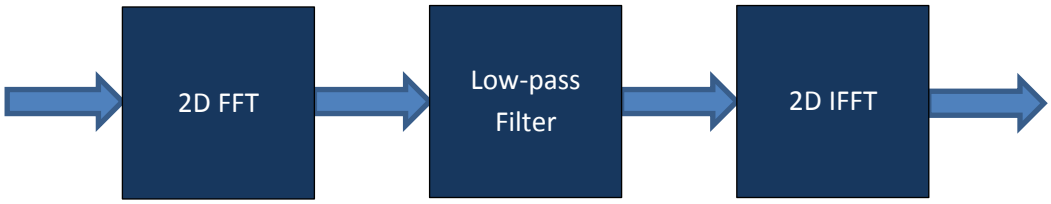

Fig 1: Image Enhancement pipeline

Fourier Transform

The standard algorithm for transforming data from spatial to frequency domain is the Fourier Transform. When dealing discrete data (like sampled digital data), then we use the Discrete Fourier Transform (DFT):

(1) $$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi k \frac{n}{N}} \qquad k = 0 \cdots \text{N-1}$$

The inverse transform (IDFT) is defined as:

(2) $$x_n = \frac{1}{N} \cdot \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi n \frac{k}{N}} \qquad n = 0 \cdots \text{N-1}$$

This formula involves $O(N^2)$ operations. Fortunately, a faster algorithm was invented, called Fast Fourier Transform (FFT) that performs the calculation in $O(N \cdot lg_2 N)$, or better, operations. It takes advantage of recurring terms in the DFT and avoids the re-calculations.

A two dimensional transform is performed in two steps - first transform each row using a regular FFT, then transform each column with a regular FFT. With some computer architecture it is more efficient to work on rows than on columns, so instead of the above method, one can transpose the array (an operation a.k.a corner-turn) and perform the 2^{nd} pass on rows as well, then transpose back to the original orientation.

Two flavors of the algorithm exist - Decimation in Time (DIT) and Decimation in Frequency (DIF). These are basically the same calculations in a different order, so the choice among the two is merely a matter of preference. Nevertheless, they both involve two separate operations - the $lg_2 N$ stages of FFT butterflies and the array reordering according to bit-reversed addressing.

There are plenty of sources describing the FFT algorithm available for further reading. For this work, we implemented the Radix-2 FFT variant.

Implementation

We implemented the Image Enhancement program on a 16-cores E16G3 Epiphany processor on an EMEK3 system in a host-accelerator model. The cores have 32KB of per-core SRAM. It can also transparently access all of the on-chip memory and off-chip DRAM. For this problem we decided to implement the program in a data-flow accelerator style. Here, the Epiphany acts like an image processing chip in a system with no accessible DRAM. This means that all image data and twiddle factors have to be stored locally on the per-core memory. For simplicity, we used floating point data.

Forward 2D Transform

When arranging the data arrays on the on-chip memory, there are a few possible schemes. Probably the most obvious is to divide the image to square tiles, each of size ¼ of the total height and width, and locate each tile on a core, as seen in Fig.2. We can then perform FFT on image rows and then on columns, but we need to do it in a multicore fashion, spread over 4 cores. Examining the DFT equation (and the FFT algorithm), we can see that every frequency component involves data from all the spatial domain samples. This means that we need to swap data between the four cores in order to complete the full FFT calculation.

Alternatively, we arranged the 16 cores in a column, as seen in Fig.3, such that each core handles full image rows:

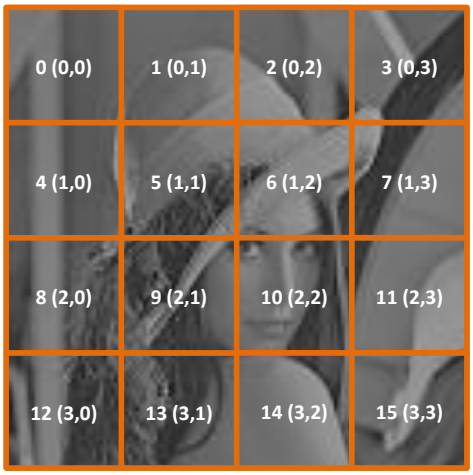

Fig 2: Matrix tiling topology of the memory

Fig 3: Column tiling topology of the memory

Arranging the cores in this way requires us to transpose the image data after the row-wise transformation, such that the columns become rows and we can perform the column-wise transform using a single core program. Then, we transpose it again to have the columns back in place.

Low Pass Filter

For the lowpass filter we used the ideal step-shaped window function, eliminating the high frequency domain coefficients, which are located around the center lines of the transformed image (which correspond to the Nyquist frequency). The filter coefficients can be seen in Fig.4:

1	0	0	1
0	0	0	0
0	0	0	0
1	0	0	1

Fig 4: 2D Low Pass filter coefficients

So, for every non-zero mask, we leave the frequency component unchanged. For every zero mask we replace the component with 0.

Inverse Transform

Examining the IFFT equation, we see that each sample is normalized by 1/N. This is required in order to satisfy the condition that $x = IFFT(FFT(x))$. Furthermore, the twiddle factors of the IFFT are the complex conjugates of the FFT. In order to reuse the same function for FFT and IFFT, we perform the 1/N normalization in the filter function. We also extend the twiddle factors look-up table to include the conjugates.

Memory Arrangement

The E16G3's per-core memory is comprised of four 8KB banks, totaling 32KB. Reading and writing from/to different banks can be performed with no penalty. For our implementation, involving corner-turn operations, we need to allocate two buffers for the concurrent swap of data between the cores. The final layout is as follows:

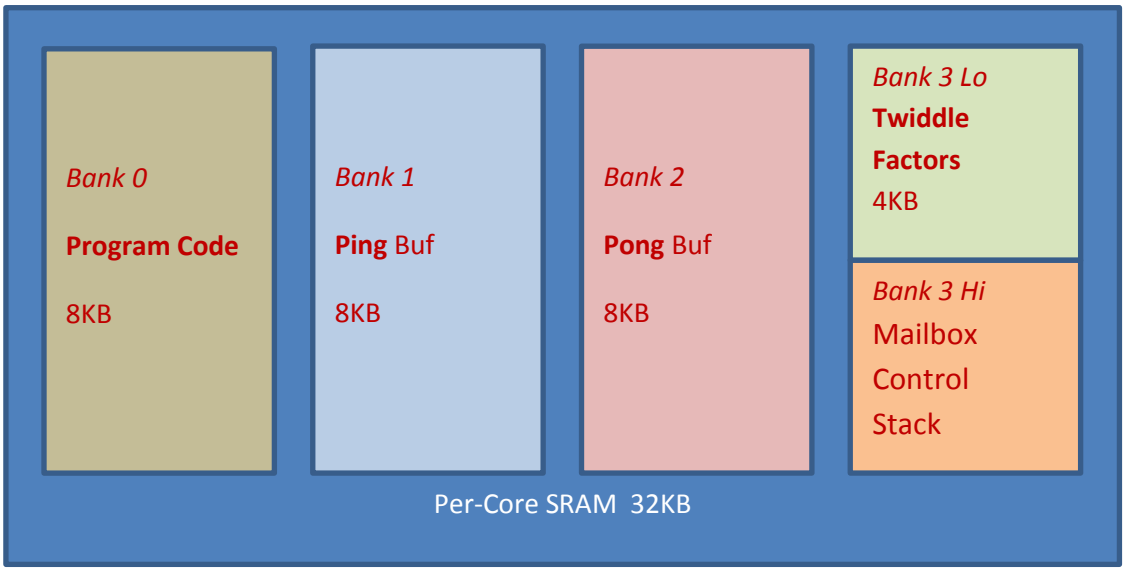

Fig 5: Per-core memory arrangement

The amount of available memory can contain an input array of up to 128×128 complex elements. For this program, we use complex type to store the pixel data. We assume that any required type conversion is done outside the FFT accelerator. Given 16 cores, each core stores 8 rows of 128 pixels.

Scalability

The program was implemented in a scalable fashion. The distribution of work across the cores is even. This implies that the time required to complete the calculation is roughly inversely proportional to the number of cores. Thus, a 64-core chip like E64G4 will complete the task in about ¼ of the time. Increasing the chip size also adds memory and thus enables the increase of the frame size. Considering the chosen memory allocation scheme, it can be seen that maximum size of the frame that can be processed is:

(3) $$S_{frame} = \sqrt{\frac{N_{cores} \times S_{bank}}{S_{complex}}}$$

Where:

S_{frame} - Witdh and Height of frame
N_{cores} - Number of cores in chip
S_{bank} - Size of a memory bank in bytes
$S_{complex}$ - Size in bytes of complex-float type (= 8)

For the E16G3 this means a maximum frame of 128×128. For the E64G4 the calculation gives 256×256. The E64G4 will process a 256×256 frame at the same time as the E16G3 processes a 128×128 frame.

In order to process a 1,024×1,024 frame, we would thus need 1,024 cores (with 8KB per memory bank). This means a chip size of 32×32 cores. The Epiphany architecture enables a virtually zero effort in arranging a cluster of chips on a single board. In this case, the chips form a glueless epiphany mesh, which means that existing code need not change when scaling the available hardware. Thus, one can arrange an array of 8×8 E16G3 chips or 4×4 E64G4 chips to form a 1,024×1,024 2D FFT engine.

Results and Analysis

The program was implemented in C and tested on the EMEK3 platform. The EMEK3 kit is based on the E16G3 Epiphany chip which has 16-cores. The chip clock frequency is set to 400MHz but may be increased to 1GHz. We tested a sample 128×128 image, to which we added some high frequency noise. The noisy image and the filtered out result can be seen below:

Noisy image

*Filtered imag*e

Measurements of the performance show that 2D FFT operation requires 0.73 msec = 294 Kcycles to complete. Of which, about 89% go to the butterfly stages, 6.5% are consumed by the bit-reverse reordering and the rest is spent on the two corner-turns. The whole image processing operation takes about 1.5 msec. This includes 2D FFT, low-pass filter and 2D IFFT.

Stage	***Cycles [x1000]***	***%***
butterflies	264	89.5
reordering	19	6.5
corner-turn	13	4.2
Total $^{(*)}$	***294***	***100***

(*) note that the total count does not equal the simple sum of the components b/c some operations are performed in parallel across multiple cores.

A more aggressively optimized, Radix-4 FFT code was implemented in another context and shows that an improvement of up to 2X to the above measurement is feasible.

Conclusions

In this paper we presented a sample frequency domain image enhancement application using 2D FFT operation. This implementation does not use DRAM for storing data and is limited by the amount of on-chip memory. We chose the cores arrangement topology so that no parallel 1D FFT implementation was required. Thus, we simplified the implementation.

The implementation is easily scalable and the performance is inversely proportional to the chip size.

Performance measurements show that the Epiphany's Netwok-on-Chip (NOC) is extremely efficient in transferring data among the cores, and that the corner-turn operation consumes a small fraction of the total calculation time.

The timing shows that using the 16-cores Epiphany, scaled to max frequency, it is possible to process thousands of frames per second.

References

[1] The program's code, in the form of an Eclipse project "fft2d" is available for download from the Adapteva website at http://www.adapteva.com/support/examples/