Abstract:

In data IO intensive workloads such as image processing and CNN, it is being increasingly pivotal to increase data throughput and enhance IO delay by increasing the cache locality. However, physical cache's performance is limited by its real-life physical constraints, and researchers have been exploring non-conventional methods to increasing cache locality. One of the methods to achieve better special locality is by Approximate Cache Access methods, exemplified by the "Bunker Cache" which cache controller can exploit the redundancy nature of the image data and map the vertical 'lines' of memory into the same cache address. In that case, whenever the processor wants to access the data 'line' of a cached data adjacent in 2D space, the cache access would become a hit and returns the same data that have just accessed before. One of the drawbacks of the bunker cache was that some extend of ISA level support was required. One of the parameters in the bunker cache controller was the "step size" and the project provided a proof-of-concept experiment which shows that performing a FFT calculation on the IO data-stream would extract such parameter rather effortlessly. For the project, A DSP based co-processor that work together with the Bunker Cache controller was constructed, and it is able to automatically calculate the parameters with each IO on image data from disk to memory, at the same time seamlessly and synchronously computes FFT on the data stream and updates the controller parameter registers. This co-processor is able to hide the implementation details of the Bunker Cache from the programmer and make the bunker cache more like a black box to the user.

Background

The Bunker Cache was a new cache placement scheme that can seemingly increase the spatial locality of the access pattern of a program. This scheme works is set up on the premise that the data accessed by the program is similar in storage space in a stride interval. For example, a 2D image's pixel value does not change much both horizontally and vertically. If we would tradeoff data accuracy of the image, we can map the pixel values of the "image strip" into one single location based on some mapping function, so when the program is accessing pixels under/above a former accessed location, the cache would hit again (even though it should not) hence reduce cache access frequency and improve hit rate. A sample of the access method of the bunker cache is described in Fig 1.

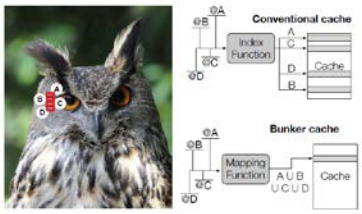


Fig 1, a bunker cache access sample, (img source J.S.Miguel et al, 2016)

As shown in Fig 1, every few adjacent image row segments were mapped into one single cache location based on a certain function. However, one of the parameters from the mapping function requires programmer's ISA-level pre-define to work properly, which is the *stride* parameter. If you imagine the image as a 2d C array, then the stride value is the row offset multiplier, AKA, the width of the array.

The basic concept of the co-processor was based on the promise of that, if we could know the *length* of an image as a 2D array, we can have a good *estimation* on the dimension (width x height) of the image by performing a Fourier Transform on the image. Hence extracting the stride parameter.

The Fourier Transform (FT) is a mathematical computation that extracts the frequency characteristic of some certain signal (formula shown in fig 2) (effect shown in fig 3). Its commonly used in audio or image processing, or anything related in frequency modulating to some signal in general. However, as shown in Fig2, the math of FT works on infinite signal with continuous sampling which is not practical as it requires infinite computational resources.

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi)\, e^{2\pi i x \xi}\, d\xi,$$

Fig 2, FT on infinite, continuous signal

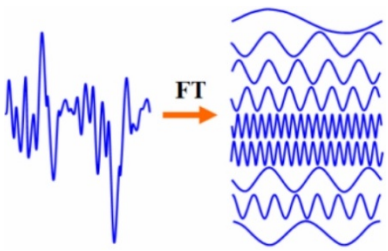

Fig 3, example of How FT extracts frequency characteristic

A more modern and digital approach on FT is called Discrete Fourier Transform (DFT), which is a variant of the original FT formula but added approximation for sampling and can work on finite and discrete data, which a computer is very much able to calculate (Fig 4). To save energy and improve, a further approximation of DFT called Fast Fourier Transform (FFT) is implemented on hardware or software which allows almost linear time complexity on its inputs.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}$$

Fig 4, formula of DFT

The reason of why performing a FT on the image can extract information on its dimension was a little unintuitive but let me try to explain. It conceptually works very much similar to the main idea of the Bunker Cache, which is based on the premise of special redundancy. Imagine we have an image, but every row is the same as its neighbor rows, which is a decent assumption in real life. Then if we flatten, or stretch out the whole image by row major, just like storing a 2D array in 1D memory space, it would be intuitively reasonable that it appears as if a single row has appeared multiple time throughout the 1D space. This repetition would show some sort of frequency on some value and could be extracted by performing a DFT on the input 1D array. In fact, if every row is exactly the same as another, then the *second* peak on the output waveform of the DFT would sit at exactly at the quantity of the height of the image (I hid some math details here to avoid the math smoke screen). Then if we know the length of the whole array, which would be implemented on hardware by counting on continuous disk memory access sequence length, a simple division can calculate the width of the image, hence getting the stride parameter.

A proof-of-concept demo was performed on Python simulation. In the simulation, the python NumPy FFF library was used to calculate the discrete Fourie transform of a row-major flattened image (fig 5) with a dimension of 510 x 340 (W x H) with data precision of 32-bit floating point representation. The result is shown in fig 6.

Fig 5, example image

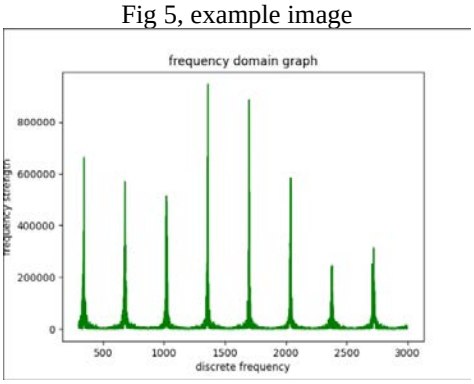


Fig 6, FFT of the owl image, flattened by row major, masked index 0-100

The first peak of the graph was masked as only the second peak's location is what we are looking for. The first peak is not chosen is because in common DFT applications, the first peak always sits at index 0, which stands for DC-offset. We can clearly see a peak at around 340, which is the same as the *height* of the image. If we flatten the image by Column major instead of row major, then the second peak of the output would be as shown in fig 7, which sits at exactly index 510.

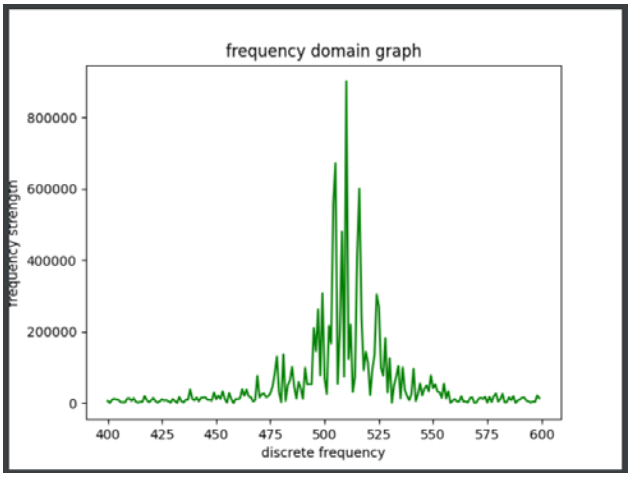


Fig 7, FFT of owl image, flattened by column major, zoomed at near Fd = 500

Results

After the python-level proof-of concept simulation, a hardware-level co-processor core with its controller was constructed on Verilog, and its behavior was simulated on Verilog. And I then Run FFT with 8-bit integer accuracy on some file data stream to see if the correct output peak could be captured.

I choose 2 images to perform the test, the first was the owl image showed in fig 5, another image is a more mathematically more confusing image which is a zebra (fig 8). Both images were converted into greyscale for simplicity, and their row-major data sequence was fed on the co-processor, which in real-application would be probing on the data bus continuously.

Fig 8, test image (dim: 600 x 400)

For the owl image with a height of 340, the co-processor determined its image height to be 387 as shown in fig 9, with the red waveform as its FFT output, and the 'counter' wave represents the FFT index. We can see the peak is at around index #387

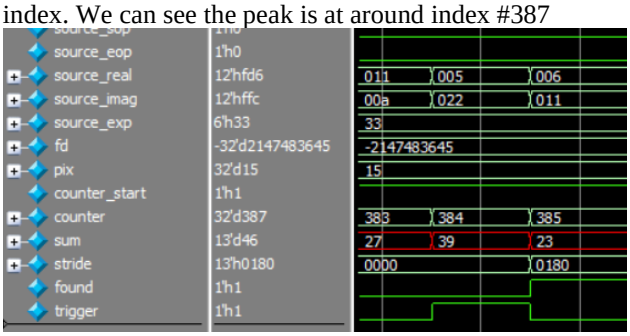


Fig 8, HDL simulation waveform

Similarly, for the more confusing zebra's image, which have a height of 400, the HDL simulation gives out the estimate height of 327.

Discussion

After demonstrating that this co-processor core 'kind of' works, there are still some problems must be addressed. The first one is that Why the result of HDL simulation differs so far from the software-based simulation? I think the answer is to blame the data accuracy. For the software simulation, a data accuracy of 32-bit floating point was used, while for the HDL simulation the core was only written for 8-bit fixed point representation. This lack of data accuracy was the main source of the significant offset from the estimated image dimension to the actual parameter.

The second elephant in the rooms was that: Why the zebra's image has a more error than the owl one? The zebra's image was cherry picked to demonstrate this potential error. The answer is from the inaccurate premise the whole idea was build upon. The premise was to assume that every row differs only slightly and randomly from the other rows so that it appears repetitive in 1D space so the FFT can identify the major frequency. However, the zebra image have many diagonal strips. This adds a huge offset on the frequency domain as diagonal strip would shift the FFT's computation (math omitted).

And the final yet most important question is that: Would this method work in real life. Sadly my answer is: highly unlikely. This is because the result of the co-processor diverges so far from the actual image parameters that the access becomes very accurate in column space. Here's what a access pattern looks like if we only change the 'row multiplier' of a C-like 2D array only by 1 (fig 9):

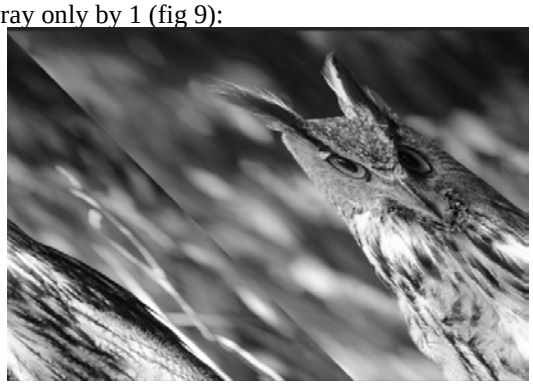

Fig9, accessed image with 'stride' 1 unit smaller than it should be

To some special error tolerant application, such as image color statistics, or histogram computation, this false in space would be acceptable. But for the vast majority of other application such as image recognition this could leave a huge error on the result.

Conclusion and future work

So in conclusion, the fix-point representation based Bunker Cache was not able to deliver an accurate parameter for the bunker cache, this comes from one major reason and several minor reasons. The main reason for the error was because of the data representation issue, that the fixed-point computation was not accurate enough to locate the DFT peak on the input data stream; on top of that, image's diagonal

component, as well as the approximate computation nature of FFT would also contribute to the error. Some of the future improvements would be adding floating-point support, as well as fine-tuning the code to increase accuracy.

All project files: Bald-Badger/bunker_FFT (github.com)

Python simulation: ~/FFT_sample

All HDL code and simulation: ~/FFT/quartus