**A Comparative Analysis of Implementing a Deconvolution of a Diffused Image in real-time as a front-end for Calcium Imaging**

Ray Duran  
Department of Electrical Engineering and Computer Science  
Univertisy of North Dakotaray.duran@und.edu

*Abstract* - **There have been many advancements in miniaturized microscopes that produce in-vivo calcium extraction imaging in the study of neuroscience. Typically these small form factor cameras use small lenses but are restricted by the normal physical tradeoffs of resolution and field of view. In place of small lenses there has also been a lot of development for on-chip fluorescence microscopy using diffuser lenses that can allow for better tradeoffs. While diffuser lenses provide a low-cost and simple alternative to traditional lenses, they do require the solution of an ill-posed inverse problem that for in-vivo calcium extraction require real-time processing. We aim to provide in this paper a comprehensive comparison between the latency computation time using a gradient descent optimization minimizing a convex function for three processing devices, a Graphical Processing Unit (GPU), Central Processing Unit (CPU), and Field Programmable Gate Array (FPGA).**

I. Background

With the acknowledgement of the Nobel Prize in Chemistry in 2008 for Shimomura, Chalfie, and Tsien for the discovery and development of the green fluorescent protein GFP, the past decade has touched off a stream of advances and papers in the extraction of Ca2+ in-vivo imaging of large number of neural cells. At its’ core calcium imaging, Fig. 1, requires the processing of four major areas: background filtering, motion correction, neural enhancement and finally calcium trace extraction for processing [1][2]. On way to reduce the dimensionality of the data and provide noise reduction is by the ubiquitous principal component analysis (PCA). As a first step this filtered data would then be fed into the next stage of our calcium image processing, which would usually entail a way to identify the actual neurons themselves. One method tried by [3]. was to use Independent Component Analysis, which used the FastICA [4], which would be of aid in the motion correction portion of the processing to help compensate for any motion of the brain. Finally, ideas from

image processing, involving image segmentation could help with the final extract.

Diagram

Description automatically generated

Fig. 1 Calcium Imaging Flow

While calcium imaging defines the approach technique with which we intend to study neural activity, it remains how best to gather the light field, fluorescence light, considering the maximum field of view, resolution, and efficiency in collecting light. Here, we investigate the use of diffusers as our front-end system for a calcium imaging

Diffusers provide a light field representation of 4-D information into a 2-D format, and in the process capture spatial and angle information. While in this paper diffusers provide our principal optical component, they have been successfully used in many other application (3D neural activity [5], compressive radar imaging [6], synthetic aperture [7], and visual odometry [8].). As a highly diffractive medium, the speckles (caustic patterns) from interference in our diffuser create a type of signature that can encode plenoptic information about our illuminated object and with proper assumptions allows us linearity in reconstruction that lends itself to well-established inverse problem definition and optimization techniques. To aid us in solving diffuser inverse problems with less than full rank we make use of the randomness they provide is to see if this fits within the theoretical framework of compressive sensing. Mathematically, if the combination of the measurement and representation matrix satisfy the property of mutual coherence then full reconstruction of the sparse system is possible.

Graphical user interface

Description automatically generated The literature of using diffusers for lens less applications has been explored [9] [10]. In addition to their diffusers, amplitude masks have been studied [11]. Finally there are other ways of approaching the reconstruction problem that do not entail, one example is the study of phase space itself, of light as it passes through a scattering medium [12].

Having briefly mentioned the overall system modality, calcium imaging, and our optical component choice, diffusers, we now outline the three technologies we intend to compare for the solving of our inverse problem: Graphical Processing Unit (GPU), Central Processing Unit (CPU), and Field Programmable Gate Array (FPGA). Here, it is best to investigate these three computing paradigms by studying three aspects of computing: 1. The programming model: As the language that these computing engines use highlights their internal architecture and will allow us to gain insight into how latency can be computed. 2. How memory is partitioned and used in the device. And 3., how we achieve concurrency in our computation.

Table

Description automatically generated For GPUs, Fig 2, we first look at the programming model, and here we will use an Nvidia device, that uses a language called CUDA, that stands for Compute Unified Device Architecture (CUDA) [13], which is just an extension of the C/C++ language, with keywords used by the NVIDIA processor. The CUDA code breaks down the programming workflow into three steps: i) Allocating and internalizing data. This means since a CPU is usually the starting point of a program that the CPU copies data to be used by the GPU appropriate memory. ii) Calls the GPU kernel and launches threads. iii) Upon completion of computing task, copies data back into the CPU memory. Memory usage in the NVIDIA GPU consist of shared DRAM with another computing element. Concurrency in GPUs is achieved by the architecture and a programming construct. GPUs group threads in a hierarchy called warps. Here, threads are controlled intrinsically to make sure that there are no data hazards to computing, A programming construct called barrier synchronization allows

Fig. 2. General GPU architecture

synchronization across warps that pauses all threads in a block when needed.

For the technology of CPUs, Fig 3, we choose to compare the 12th generation Intel i7 multicore processors [14]. CPUs are programmed by C/C++ and often can include libraries that are needed for the math of linear algebra such as Blaze, GNU scientific library [15]. Memory in CPUs can be local cache or external DDR. Concurrency in CPUs is achieved with the use of multicores, of which in our comparison the i7 Intel processor has four.

Fig. 3. General CPU architecture

Field Programmable Gate Arrays (FPGAs), Fig. 4, are dominated by two companies, AMD (Xilinx FPGAs) and Intel (Altera FPGAs). Both families of FPGAs are ubiquitous in the embedded comm are widely used in embedded products that require speed (MHz), latency(time) and allow for the flexibility of being able to re-program an algorithm even when the product is deployed in the field. Here is a small sampling of the use of FPGAs in systems [16][17][18][19].

FPGAs use a family of languages called Hardware Description Language (HDL), of which there are two main dialects, VHSIC Hardware Description Language (VHDL) and Verilog, with System Verilog a newer version of Verilog. HDL allows capture of a design down to a level called Register Transfer Logic (RTL) which breaks down the computation into fundamental digital logic elements such as flip-flops(registers), state-machines, shifters, muxes, adders, AND and OR gates, etc. This flexibility of course permits almost any kind of computation and data transfer with the only limitations being speed and resources of the device. The memory of the FPGA exists throughout the device and can therefore be allocated as shared or global memory depending on the exact application. Additionally, modern FPGAs also consist of interfaces, DDR memory as one example, that allow for the expanse of external memory, thus greatly enlarging the memory available for logic to process. Finally, since an FPGA is essentially a sea of gates that operate with many clocks, concurrency is implicit as a result.

Diagram

Description automatically generatedFig. 4. General FPGA Architecture

Text

Description automatically generatedII. Methods

In this section below, we present a variation of the gradient descent algorithm called the Fast Iterative Shrinkage-Thresholding Algorithm (FISTA), Fig 5. Here we assume that our calibrated point spread function is invariant, Fig. 6., and that allows us to reconstruct the image using linear deconvolution.

A picture containing text, gallery, different, clock

Description automatically generatedDiagram

Description automatically generated

Fig 5. FISTA Overview Reconstruction

Text

Description automatically generatedIn theory a lens less system has all the information available at the sensor. However, the problem is ill-posed. Introducing a diffuser gives structure to our point spread function, or random matrix, allowing us to solve an otherwise intractable problem. If we enforce sparsity as a prior, and non-negativity (no negative pixels), minimize for the least squares we are trying to solve the equation below

Text

Description automatically generated Here in (1) we do not show the regularization term that helps enforce sparsity, instead we want to show the main thrust of how we approach minimization overall

A screenshot of a computer

Description automatically generated with low confidence Here, the significance of (4) is that we have reduced the problem of computing A which could be a very large matrix to the adjunct of M and C, a much more manageable problem. (C is introduced as a cropping matrix)

Here there are a few points to note. The first is that the M matrix in (5) has already been decoupled from A, and this is because if we included the cropping effect into A, the matrix would be ill-conditioned. Continuing with (5), we have just expressed the convolution of “Mv” as a product of Fourier and the inverse of that dot product but have expressed the equation as products of matrices.

Text

Description automatically generatedHere we substituted (7) back into (4) to give us (8) and taken the adjunct of (8) to give us (9). At this point (8) and (9) we can implement into code to solve our iteration (10).

Fig. 6. PSF is invariant.

Fig. 7. Shows the PSF and response of target of interest and how the FISTA as it operates would iterate on our original diffused images. (Fig. 8.)

A picture containing text, monitor, electronics, display

Description automatically generated

Fig. 7. From Liang , UND optics lab diffuser. PSF (left) and capture response of object(right)

Graphical user interface

Description automatically generated with low confidence

Fig. 8. From Liang , UND optics lab diffuser. Upper Left is for 20 iterations reconstructions. Upper right, Lower left, and Lower right are 40,60 and 80 iterations of FISTA algorithm

**III. System Implementation**

Diagram

Description automatically generated To implement the FISTA algorithm from our method section II into our different computing platforms (CP): GPU, CPU and FPGA, we must consider the structural differences of each CP. The main components of our algorithm consist of six major sections: i) padding/down sampling, ii) FFT/IFFT, iii) Pre-computations of convolutional matrix and adjoint iv) Cropping, v) gradient-descent updates, vi) Data flow processing of new images into and out of memory, Fig. 9.

Fig. 9. FISTA data flow

Here owing to the architecture of the GPU, we can define the threads in our blocks and grids as (2D) to process our 2D FFT/IFFT functions. Data movement in an out of the GPU is usually assisted by one-board CPUs that act as direct memory access (DMA) controllers. The rest of the FISTA functions, like down-sampling, padding, and cropping can be implemented in threads Table 1. shows the possible specifications from a GPU that we can use.

Table

Description automatically generated with low confidence

Table 1 [20] GPU Specifications

Owing to the obfuscation of our C/C++ implementation of the FISTA algorithm by the C compiler, where the scheduling of the cores and threads is handled at the OS level, the practice of extracting maximum concurrency in operation can be challenging. Here the i7 FPU will handle the DSP elements of our algorithm, with the assistance of hyperthreading. This includes both FFT and IFFT operations in our FISTA algorithm along with multiplies in the Fourier domain along with any additions and subtractions that are required from the gradient update operations [21][22]. Padding and down-sampling and cropping can be burst operations and are repetitive and can be instruction cached. Finally, when possible, the pre-computed convolutional matrix and adjoint matrix can be data cached and the high-speed external DDR4 memory can be used for data movement of new diffused images (See Table 2)

Platform Intel Core i7-8700 CPU @

Processor Architecture i7

CPU cores 4

Memory Size 32.0GB

Table 2. CPU Specifications

In an FPGA, using fixed point 16 bits, with flexibility offered by the fine-grain architecture of logic, memory, and DSP elements we breakdown our FISTA algorithm as follows. We buffer up the several frames of our original image that received from our CMOS sensor of 3280(H) x 2464(V) into external DDR4 @ 32 bits wide. In the next step we would down sample our original image, this would help us save in total computational time, into an image of size 256x324 pixels. Down sampling in digital signal processing is not an intensive operation and just really involves discarding N samples from a row or column. Proceeding, we would next need to pad our data so that we can streamline the processing into an FFT, as most FFTs are radix 2. So, for our FPGA implementation we would then convert our 256x324 image into a 512x1024 pixel image. While FFT cores of radix 2 and N size up to 4K are just IP cores of an FPGA vendor, 2-D cores need to be constructed out of those libraries in general. Here in our FPGA implementation, we would use internal block ram to buffer for the transposition of row FFT into our column FFT data flow. With the computation of the FFT, we multiply our pre-computed convolution matrix before

computing the inverse FFT. We then performing the cropping operation in the time-domain where we again truncate to a 256x354 pixel image. At this point we have the convolution matrix “A” matrix multiplied by our first estimate of the recovered image “v”. Proceeding, now with the first step of the gradient subtraction operation. All the FPGA processing post the FFT do not require heavy operations and can be done with high-speed and low latency in DSP multiplier blocks where we use the arithmetic component of the DSP. Having up to this point effectively half of the processing for one image, we repeat the process from padding to output of inverse FFT, but now use the adjoint convolution matrix as our constant multiplicative factor. Finally, we stream this updated prediction of our reconstruction back into memory to be repeated all over again by all the elements above. Based in some Python experimentation with the FISTA algorithm we estimate that we would have to repeat the process above approximately 50-100 times for a reasonably valid image of the deconvolution.

FPGA Architecture Kintex Ultrascale +

System Logic 475K logic cells

DSP Slices 1,824

Memory 34.9Mb

GTY 32.75 Gb/s Transceivers 16

I/O 304 pins

DDR4 32 bits

Table 3 FPGA Specifications

**IV. Results**

The frequency of the operations is repeated for 100 iterations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operation | Size | Frequency | Portion of Code (Note 6) | Cycles/Time  (Notes 1,3) |
| Normalization | 256x324 | (Note 10) | Pre-processing raw data | 1 (Note 4) |
| Down Sample | 2048x2592 | (Note 10) | Pre-processing raw data | 1 (Note 4) |
| FFT/IFFT | 512x1024 | 4 | During gradient update | (Note 9) |
| Pad/Crop | 512x1024 | 3 | During gradient update | 1 (Note 4) |
| Add/Sub | 512x1024 | 1 | vk – alpha\*gradient (Note 5)  xk+((tk-1/t\_k1)\*(xk-x\_k1) | 1 (Note 4)  1 (Note 4) |
| Sqrt | N/A | 2 | N/A (Note 7) | N/A |
| Mult (element-wise) | 512x1024  512x1024  512x1024 | 1  1  1 | vk – alpha\*gradient  H.\*vk  H\_adj.\*X | 1 (Note 2)  1 (Note 2)  1 (Note 2) |
| Square | N/A | N/A | N/A (Note 7) | N/A |
| Divide | N/A | N/A | N/A (Note 7) | N/A |
| Non-negativity | 512x1024 | 1 | during update of xk | 1 (Note 8) |
| Table 4 Summary of Operations.  Notes/Assumptions: 1) GPU[23] and FPGA takes 1 clock cycle. 2) CPU takes 3 clock cycles[24] 3) All cycles are evaluated at 300 MHz which is a reasonable speed for off- core logic. 4) Normalization, Down sampling, pad/crop and adds takes 1 clock cycle 5) Ignore calculation of alpha term. 6) Ignore initialization of PSF matrices in calculation 7) For scalar calculations we ignore computation because of negligible size compared to matrices.8) non-negativity assume one clock 9) See FFT/IFFT Equations for calculating time of computation. 10) The frequency of the operations is repeated for 100 iterations except for Normalization and Down sample which are done once for block. | | | | |

Here are the FFT calculations that we referenced in table 4 above. Note that we are not including any of the data transfer times for CPU calculation. Our reasoning for this is that the GPU requires this as an extra step for processing and that data transfer times for the CPU and FPGA can be “baked into” the calculations that we made for the down sampling. In other words we just say that we are preforming the fetches of data from memory and that that counts as our data transfer.

Now, for the CPU and GPU FFT we compute, with Mtr=Ktr=Ctr, n=512, m = 1024, extracting from table 1.

**FFT specific System Parameters**

Ktr = 1.11e-8; % Matrix transpose on CPU

KgMa = 5.95e-10; % Memory allocation on GPU

Kc2g = 6.16e-9; % Data transmission from CPU to GPU

Kg2c = 5.68e-9; % Data transmission from CPU to GPU

KgP = 2.73e-11; % Pre-processing of 1D-FFT on GPU

KdDP = 7.28e-12; % Post-processing of 1D-FFT on GPU

KcFFT = 2.61e-9; % 1D-FFT on CPU

KgFFT = 1.65e-10; % 1D-FFT on GPU

KgMF = 3.01e-9; % Memory releasing on GPU

**2D FFT computational time**

CPU-only

r = 0;

C\_col = (1-r)\*n\*m\*2\*Ktr\*Ctr + (1-r)\*m\*n\*log(m)\*KcFFT;

C\_row = (1-r)\*m\*n\*log(n)\*KcFFT;

time\_cpu = C\_col + C\_row;

GPU-only

r = 1;

G\_col = r\*n\*m\*2\*Ktr\*Ctr + r\*m\*n\*log(m)\*KgFFT + ... r\*n\*m\*(KgMa + Kc2g + KgP + KdDP + Kg2c );

G\_row = r\*m\*n\*log(n)\*KgFFT + ... r\*n\*m\*(Kc2g + KgP + KdDP + Kg2c + KgMF );

time\_gpu = G\_col + G\_row;

For FPGA FFT times we use reference [25], where latency is 156us for a 32K FFT much larger than we need for our 512-point matrix. So, as an approximation for the FFT we ignore the transposition( because our FFT point size is much larger than our FFT) and we double up the value for the 2-D. This gives us 312 us time for the FPGA 2-D FFT. Now computing with frequency = 300e6, cycle\_time = 1/freq, time\_fpga =312e-6,iter= 50,n=512,m=1024

**System GPU/CPU/FPGA Computations**

padding = n\*m\*cycle\_time\*3;

add\_sub = n\*m\*cycle\_time\*3;

mult\_gpu = n\*m\*cycle\_time\*3;

mult\_fpga = n\*m\*cycle\_time\*3;

mult\_cpu = n\*m\*cycle\_time\*3\*3;

non\_neg = n\*m\*cycle\_time;

**Total times for one iter:**

gpu\_one\_iter\_time = time\_gpu\*4 + padding + add\_sub + mult\_gpu + non\_neg;

cpu\_one\_iter\_time = time\_cpu\*4 + padding + add\_sub + mult\_cpu + non\_neg;

fpga\_one\_iter\_time = time\_fpga\*4 + padding + add\_sub + mult\_fpga + non\_neg;

**Total times for reconstruction:**

gpu\_total\_time = gpu\_one\_iter\_time\*iter;

cpu\_total\_time = cpu\_one\_iter\_time\*iter;

fpga\_total\_time = fpga\_one\_iter\_time\*iter;

Applying the FFT system parameters, 2D FFT computation, and System GPU/CPU/FPGA computations gives us:

|  |  |
| --- | --- |
| Computing Device | Frames/Sec |
| GPU | .33 Hz |
| CPU | .20 Hz |
| FPGA | 1 Hz |

Table 5. Comparison Results

V. Discussion

Here in the discussion section, we briefly state some ancillary thoughts: number of iterations, single vs double, and mix-matching processors. The first of these, we assumed that we could get a decent iteration at 50Hz, this is an assumption, and the number may need to be higher. As far as single precision vs double, although we use double precision for our Python baseline code, we may be able to use single for our calculations. And finally, in this comparative study we only used each processor in a standalone mode. It may very likely be that a mix and match of GPU with CPU and offloading to an FPGA is a more optimized latency approach and something seen often in the embedded computing world.

VI. Conclusion

Using Python 3.1 and running the Python IDE on our i7, table 2, we get a processing time of 97 seconds for an operating frequency of .01 Hz. So, from our calculations we predict at least 100 times speed up. Of course, we would like to do better and approach video rates, 30 Hz. This may involve a faster way to do optimization with an algorithm such as alternating direction method of multipliers (ADMM).

Future work could also include incorporating calcium image and wireless processing. Finally, all of this is just information in one direction, a more advanced system could also include optogenetics control of a system with a closed loop.

VII.References

1. A. Giovannucci et al., “ OnACID: Online Analysis of Calcium Imaging Data in Real Time,” 31st Conference on Neural Information Processing Systems (2017)
2. J. Lu, et.al., “MIN1PIPE: A Microscope 1-Photon-Based Calcium Imaging Signal Extraction Pipleine,” Cell Reports 23, 3673-3684 (2018)
3. E. Mukamel, A. Nimmerjahn, M. Schnitzer, “ Automated Analysis of Cellular Signals from Large-Scale Calcium Imaging Data, “ Neuron 63, 747-760, (2009)
4. A. Hyvarinen, J. Karhunen, “ Independent Component Analysis, “ Wiley Inter-sceince (2001)
5. N.C. Pegard et al., “Compressive light-field microscopy for 3D neural activity recording,” Optica 3, 517-524 (2016).
6. R. Baraniuk, P.Steeghs, “Compressive radar imaging,” IEEE Radar Conference (2007).
7. V. Vaish, et al., “Reconstructing occluded surfaces using synthetic apertures: Stereo, focus and robust measures,” Proc. 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (IEEE, New York, 2006).
8. N. Zeller, F. Quint, and U. Stilla, “From the calibration of a light-field camera to direct plenoptic odometry,” IEEE J. Sel Top. Signal Process 11, 1004-1019 (2017).
9. Z. Cai, et al., “ Lenless light-field imaging through diffuser encoding,” Light :n Science & Applications(2020)
10. N.Antipa, et al., “DiffuserCam: lensless single-exposure 3D imaging,” Optica (2017)
11. J. Adams, et al., “ Single-frame 3D fluorescence microscopy with ultraminiature lensless Flatscope,”Sci. Adv,3 (2017)
12. H. Liu, “ Optical Phase Space Measurements and Applications to 3D Imaging and Light Scattering, “ Phd Thesis , U.C. Berkeley, (2018)
13. D. Luebke, “Cuda: Scalable parallel programming for high-performance scientific computing,” in *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to* *Macro*, 2008, pp. 836–838.
14. <https://www.intel.com/content/www/us/en/products/sku/227853/intel-core-i71265ue-processor-12m-cache-up-to-4-70-ghz/specifications.html>
15. <https://en.wikipedia.org/wiki/Comparison_of_linear_algebra_libraries>
16. J. Kim, “ Hardware Accelerator Systems for Artificial Intelligence and Machine Learning,” Advances in Computers, (2001)
17. V. Arkalgud, S. Aziz, “ Hardware Implementation of LDPC Decoders,” Rerource Efficient LDPC Decoders, (2018)
18. R.C. Cofer, B. Harding, “ Advanced Interconnect,” Rapid System Prototyping with FPGAs, (2016)
19. F. Mehidipour, et al., “ Energy Efficiency in Data Centers and Clouds,”Advances in Computers, (2016)
20. R. Lacouture, “ GPUBLQMR: GPU-Accelerated Sparse Block Quasi-Minimum Residual Linear Solver,” Master Thesis, Northeastern University, (2021)
21. Y. Ogata, et al., “An Efficient, Model-Based CPU-GPU Heterogeneous FFT Library,” IEEE Explore, (2008)
22. M. Frigo and S. G. Johnson. The design and implementation of FFTW3. In *Proceedings of the IEEE: Special issue* *on Program Generation, Optimization, and Platform Adaptation*, volume 93, pages 216–231, 2005.
23. S. Markidis, S. Chien,E. Laure, “ NVIDIA Tensor Core Programmability & Precision,”European Commision H2020,Grant Agreement No. 671500(SAGE0, (2018)
24. Y. Ogata, et al., “An Efficient, Model-Based CPU-GPU Heterogeneous FFT Library,” IEEE Explore, (2008)
25. K. Nguyen, J. Zheng, Y. He, and B. Shah, “ A High-Throughput , Adaptive FFT Architecture for FPGA-Based Space-Borne Data Processors, “ NASA Contract