A HARDWARE ACCELERATED APPROACH FOR IMAGING FLOW CYTOMETRY

Dajung Lee[1], *Pingfan Meng*[2], *Matthew Jacobsen*[2], *Henry Tse*[3], *Dino Di Carlo*[3], *Ryan Kastner*[2]

[1]Electrical and Computer Engineering, [2]Computer Science and Engineering
University of California, San Diego, La Jolla, CA, USA
email : {dal064, pmeng, mdjacobs, kastner}@eng.ucsd.edu
[3]California NanoSystems Institute, Los Angeles, CA, USA
[3]Department of Bioengineering, University of California, Los Angeles, Los Angeles, CA, USA
email : henrytse@ucla.edu, dicarlo@seas.ucla.edu

ABSTRACT

Imaging flow cytometry uses high-speed flows and a camera to capture morphological features of hundreds to thousands of cells per second. These morphological features can be useful to isolate sub-populations of cells for life science research and diagnostics. Our experimental setup utilizes a high speed 208×32 resolution CMOS camera, operating at over 140,000 frames per second (FPS). In each frame, the analysis routine detects the presence of an object, and performs morphology measurements. Real-time cell sorting requires a latency under 10 ms in addition to a throughput of 140,000 FPS. In this paper, we will describe GPU and FPGA accelerated implementations of the image analysis necessary for an automated cell sorting system. Our FPGA design results in a 38× speedup over software, providing 2,262 FPS with 11.9 ms of latency. Our GPU implementation shows a 22× speedup, supporting 1,318 FPS with 152 ms of latency.

1. INTRODUCTION

Imaging flow cytometry is a high throughput technique that can determine a variety of cell characteristics. It uses a microfluidic approach to uniformly deliver cells into an extensional flow region which causes high strain rates on the cell. Cell deformation provides information that can determine cell states or properties. These properties can be used for clinical diagnostics, stem cell characterization, and single-cell biophysics [1]. They are also useful for classification and sorting, e.g., mature stem cells can be separated from immature ones. However, while commercial instruments exist to record images and morphological metrics from thousands of cells in flow, no technologies can perform image analysis at sufficient speeds to sort cells at similar rates. Such a capability will enable high-purity genetic analysis of sub-populations of cells in diagnostic samples or samples of research interest.

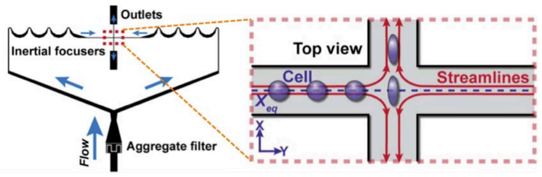


Fig. 1. Experimental setup for imaging flow cytometry. The cells enter from the bottom, proceed into inertial focusing channels, and are imaged in the extensional flow region shown shown on the right.

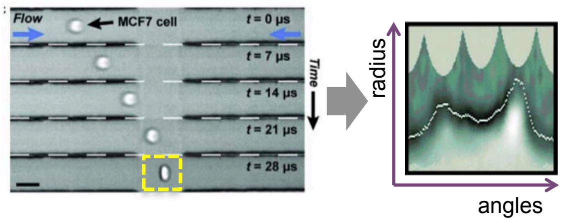


Fig. 2. Automated image analysis tracks the cell through the extensional flow region (left) and determines morphological properties, derived from the radius at each angle (right).

Our experimental setup for deformability cytometry is shown in Figure 1. A solution containing cells enters from a single inlet shown at the bottom. It splits into two directions, each of which proceeds into an inertial focusing channel. This insures that the cells are centered in the channel as they enter the extensional flow region. Cells can enter this region from either of the two directions. The convergence of the two channels creates a hydrodynamic stress on the cells in this region. Fluid must maintain a high speed to generate constant pressure on the center region. To image the cell at this speed, a microscope-mounted high speed camera captures images at 142,857 frames per second. Each frame is 208x32 pixels in resolution. The entire apparatus is

978-1-4799-0004-6/13/$31.00 ©2013 IEEE

extremely small; the channel is only 28 μm across (32 pixels in the image). Figure 2 shows a sequence of image frames as a cell moves through the extensional flow region. The cell is moving from left to right in the channel. As it reaches the center of the channel, the force from the two merging flows causes the cell to deform. It is this deformation we wish to measure. This information can be used to characterize and sort the cell. Typically cells are analyzed based on some morphological paramters, such as their initial diameter, circularity, and deformability [2]. These characteristics can be easily derived if we know the exact radius at each point of the cell membrane. This is the primary output of the image analysis. The radius is measured by bisecting the cell and then radially sweeping the bisection plane around 360° like a clock hand. An example of the output is shown on the right side of Figure 2.

The primary goal is to perform the entire cell series analysis within a latency of less than 10ms to enable real-time sorting, while the camera operates at 140,000 FPS. The sorting mechanism is performed later in the process after the cells have been imaged, e.g., it will be done in a channel connected to the outlets in Figure 1. The length of a channel and the rate of flow determine the target latency to support real-time sorting. Our target is 10 ms. This is a good tradeoff between computational feasibility and channel length.

A general purpose CPU will not support this level of performance. The algorithm takes 10 seconds in MATLAB software to analyze a single image. The same algorithm modified for higher performance and implemented in C takes 0.4 seconds. Thus, a hardware accelerated approach for this image analysis is necessary.

The primary contributions of this work are:

- Design space analysis of the image analysis technique.
- A high throughput, low latency hardware architecture implemented on an FPGA using the Xilinx Vivado high level synthesis tool.
- A comparison of the FPGA design with the code running on a GPU.

The remainder of this article is organized as follows. The next section describes related work. Then we provide a brief overview of the algorithm for cell analysis, and explain the specific hardware architecture used in FPGA design. We present experimental results in Section 5, and conclude in Section 6.

2. RELATED WORK

There is a considerable amount of biomedical imaging acceleration research in the literature. In particular, in imaging cytometry technology to analyze cells and micro particles [2, 3]. Buschke et al. present a cell analysis system for image cytometry using a DSP and FPGA [3]. Their system attempts to detect the cell and automatically process the image in the same way as ours. However, our input data are bright field images which require more complex processing than their fluorescence image using a multiphoton laser beam. Image analysis is more powerful than measuring scattered lights to extract high resolution morphological parameters of the cell. Moreover, our requirements are more challenging in terms of throughput and latency compared to their photon imaging system. Their system operates at 2.33 FPS.

Tse et al. use a GPU-based system to accelerate cell analysis [2]. It parallelizes the morphological analysis that maps a pair of images at two different coordinates and bottom-hat filters for image contrast enhancement. Their morphological analysis corresponds to our coordinate conversion. But their backward mapping conversion requires redundant memory access when searching the cell wall. Additionally, their cell analysis algorithm and approach were not optimized for FPGA implementation.

Subramanian et al. have a medical imaging system using a computed tomograph filtered backprojection algorithm. They compare the performance in different designs using C, Impulse C and VHDL [4]. However, they only investigate the validity of the HLS design by comparing it with the manual design. The result shows manual VHDL design is two times faster than Impulse C because of different pipeline depths. Our work focuses on the optimization using the HLS design. We utilize Vivado HLS for high-level synthesis and can achieve the optimal performance only with HLS. Moreover, we compare our FPGA design with a many core architecture (GPU).

3. ALGORITHM DESCRIPTION

There are four steps to perform the morphological analysis required for cell sorting. These are shown in Figure 3, and described at a high level in the following. Each module must be carefully designed in order to achieve our performance targets. In many cases, we must tradeoff accuracy for performance. For example, histogram equalization works better than image adjustment for contrast enhancement. However, it is a two-pass algorithm. One pass to make the histogram and another to equalize the image. This incurs too much latency for our design.

3.1. Blob search

The *Blob Search* module detects the cell area in the input image and converts the grayscale image into a binary image with only the cell pixels active. It proceeds by first subtracting the background from the input grayscale image to retain only cell features. Then, it converts this grayscale image into

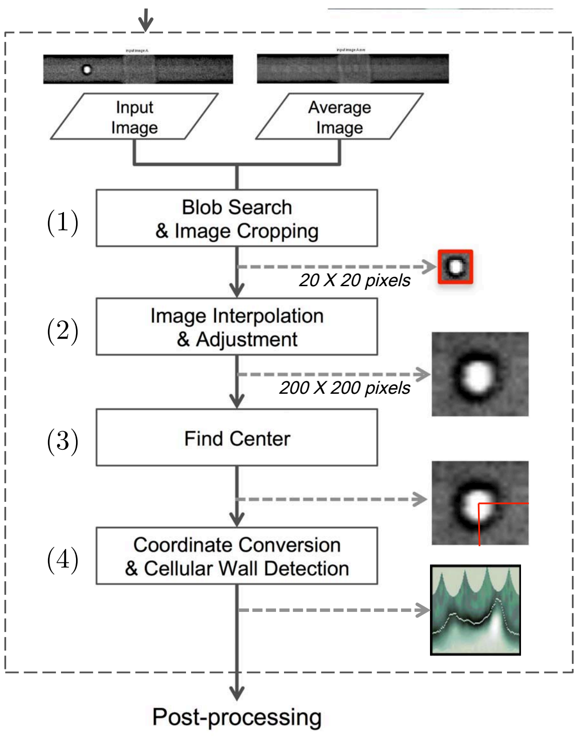

Input Image
Average Image
(1)
Blob Search & Image Cropping
20 X 20 pixels
(2)
Image Interpolation & Adjustment
200 X 200 pixels
(3)
Find Center
(4)
Coordinate Conversion & Cellular Wall Detection
Post-processing

Fig. 3. The stages of the image analysis algorithm include: (1) detecting the cell and cropping the area around it, (2) resizing the 20 $\times$ 20 cropped image into the 200 $\times$ 200 image and enhancing its contrast, (3) finding a center of the cell, (4) extracting morphological features by converting the image into polar coordinates based on the cell's center and cell walls.

a binary image using a threshold on the pixel values. Additional random noise in the binary image is removed through *dilation* and *erosion*, i.e. *opening*. The final result has only cell area pixels active (see Figure 4).

Using the binary image, the system detects the presence of a cell and its location. It creates a histogram from the image on both axes. Averaging non-zero column indices provides the approximate location of the cell. Based on this result, it crops a 20$\times$20 sized region around the cell. The entire blob search process requires a single pass per image.

3.2. Image interpolation and adjustment

To improve the fidelity of the analysis, the selected cell area from the *Blob Search* module is resized by a factor of 10. This *Interpolation* step also generates a higher contrast image by linearly adjusting the brightness level. This resized 200 $\times$ 200 image is the input to the the *Find Center* module. The outputs of this module are two images, the initial image interpolated, and the linearly adjusted image after interpolation.

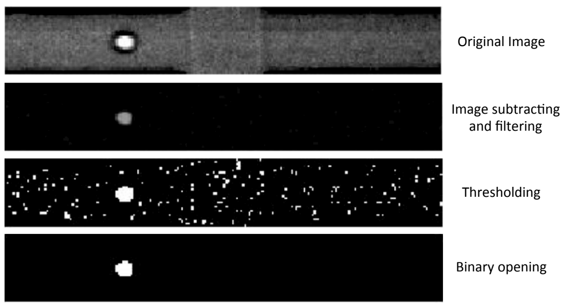


Fig. 4. The *Blob Search* module performs background subtraction, thresholding which converts it into a binary image, and opening to remove noise.

3.3. Find Center

The *Blob Search* module finds the approximate location of the cell, but the center of this window is typically not the exact center of the cell; it is often shifted slightly in the x and/or y direction. Therefore, the *Find Center* module attempts to more accurately locate the cell's center. It finds the center of the cell by converting input images into binary image and counting the number of non-zero pixels in each row and column (see Figure 5). The module processes the two output images from the Interpolation module and averages both to identify the center point. This is done to improve accuracy as specular noise can affect the results of either input. The *Find Center* module transforms these images into binary images by adaptively thresholding at different intensity values to separate the inner cell area and cell wall. It derives the candidate points for the center from four binary features, and determines the location of the cell center by averaging these points while excluding outliers.

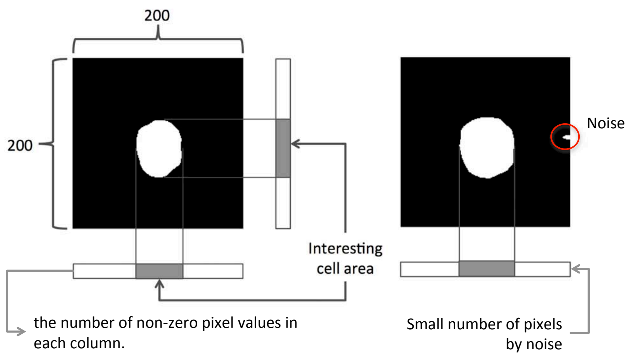

200
200
Noise
Interesting cell area
the number of non-zero pixel values in each column.
Small number of pixels by noise

Fig. 5. The *Find Center* module performs binary thresholding on the interpolated (200 $\times$ 200) image. It then counts the "positive" cell pixels in both the columns and rows restricting them to a contiguous range to avoid spurious noise. The average on both the horizontal and vertical axis defines the coordinates of the center point.

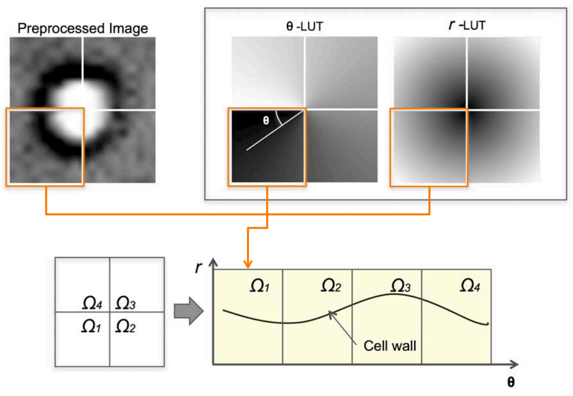


Fig. 6. The *Coordinate Conversion and Radius Extraction* module. The coordinates of the cell wall are determined by scanning through the interpolated image to find the cell wall. This coordinate is then feed into two lookup tables that provide the corresponding angle and radius.

3.4. Coordinate Conversion and Radius Extraction

Finally, the system determines morphological properties of the cell using the interpolated image and its corresponding center point. It converts the resized image from Cartesian coordinates into polar coordinates. The darkest pixels found on a line from the cell center at each angle are considered the cell wall. Figure 6 shows this process. It splits the image into four quadrants. In each quadrant, the pixels on the cell wall are determined by averaging the darkest pixels in each row. Two lookup tables provide the corresponding angle (θ) and radius of pixel for averaging.

4. HARDWARE IMPLEMENTATION

The image analysis algorithm requires careful optimization in order to reach the performance requirements for real-time cell sorting. This section provides detail on the FPGA and GPU implementations of the image analysis algorithm. In both cases, we performed design space optimization with a focus on the data dependencies and the memory access patterns of the algorithm. In particular, we discuss the execution pipeline and the Block RAM (BRAM) access optimization in the FPGA design. And we describe the thread arrangement and the memory access optimization of the GPU implementation.

Figure 7 provides a high level block diagram for the modules in the algorithm. Both FPGA and GPU implementations perform the same analysis which follows the flow in the figure. The inputs to the algorithm are the current image and the average image. The average image does not vary over time (except when changing the experimental setup) hence we consider it as a constant. The output of the analysis is radius information as shown in Figure 3.

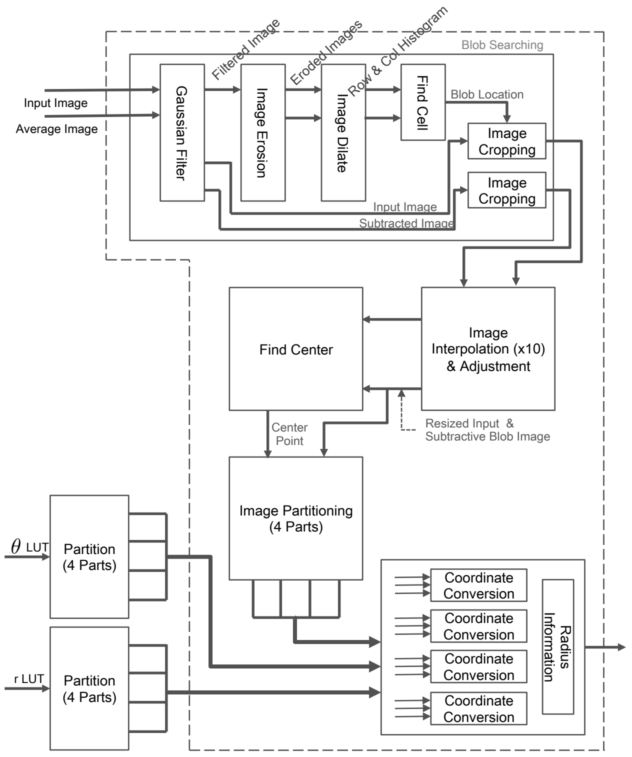


Fig. 7. The modules for the entire image analysis algorithm.

We divided the modules into three groups based upon their data access pattern and computational behavior: *Shifting Window*, *Searching*, and *Conversion*. First group includes *Gaussian Filtering*, *Image Dilation*, *Image Erosion*, and *Interpolation* modules. These all perform operations in a streaming manner on consecutive windows across the image. The *Searching* modules perform computations like finding minimum/maximum values. Modules in this group include *Find Cell* and *Find Center*. *Conversion* operations form the last group of modules. This includes *Coordinate Conversion* and other more basic operations, e.g., thresholding and subtraction, that are used within the other modules. Yet they still take a substantial amount of computational resources and time and must be carefully optimized.

4.1. FPGA Implementation

In order to achieve the required performance, we performed design space exploration using the Vivado HLS tool. We carefully analyzed each stage in the algorithm and implemented an architecture targeting the highest throughput and lowest latency. All the steps in the algorithm are designed to finish its process in one pass with minimum memory access. The remainder of this section describes the critical optimizations that we performed to achieve the highest throughput, lowest latency FPGA implementation.

4.1.1. *Shifting Window Optimizations*

A Shifting Window operation iteratively generates a consecutive window from the input image, and performs an operation across the pixels in this window. Figure 8 shows a typical windowing architecture. Here we are generating a 3×3 window. This architectures uses line buffers which are typically implemented in BRAMs. Each cycle, a read and write operation is performed on each line buffer. The incoming pixel is written into the bottom line buffer, while a 3×3 pixel array window is generated by reading the line buffers. When the current bottom line buffer fills, the top line buffer will be the new bottom. The use of the line buffers will shift in a circular fashion, as will the use of the pixel data in the 3×3 window. Pixel values are read from the window during processing. These memory cells are implemented as registers.

Generating the window can be done once per cycle. However, the window operation may take longer. For example, the *Gaussian Filter* and *Image Erosion* modules work at the rate of clock cycle per pixel. However, the *Interpolation* module takes longer primarily because the output is read into another line buffer for consumption by the next module. Since the output is larger than the input (e.g., 10× in our case), these line buffer reads are serialized.

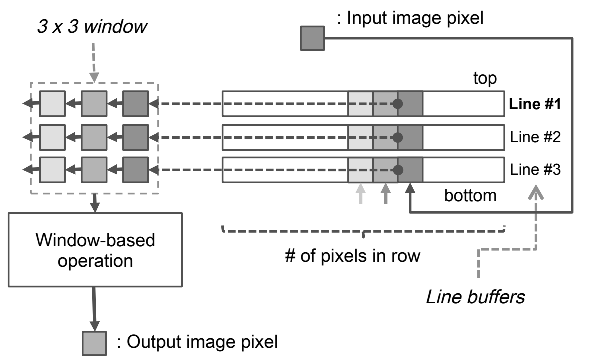


Fig. 8. The architecture design for a shifting window operation on FPGA.

4.1.2. *Searching Optimizations*

The *Find Cell* module is a searching operation that determines the approximate location of a cell in the image. It uses the *Image Dilate* module which generates a histogram array to count the number of active pixels for cell in row and column. Then the *Find Cell* module thoroughly searches this array and decides which locations should be averaged as an available cell area. The *Find Center* module works in a similar fashion; it also generates histograms for each row and column while it is performing binary thresholding on the pixel to determine if it should be considered as a cell.

The optimizations performed in these modules are mostly algorithmic. This involves changing the algorithm itself so that the operations are done in a single pass. For example, the provided MATLAB code for the *Find Center* module would first iterate across the entire image to perform thresholding. And then it would iterate over this binary image to generate the histograms. Combining these two iterations together (essentially a form of loop merging) is a simple yet extremely effective way to create a better hardware architecture.

4.1.3. *Conversion Optimizations*

We focus the discussion on *Coordinate Conversion* module since it is a major part of the image analysis algorithm. This module takes as input an image containing the cell, its center, and pixels denoting the cell walls. It converts this into polar coordinates with radius information for each angle.

We implemented this module by iterating over each pixel in the input image. We performed the Cartesian to Polar conversion for the current pixel using a lookup table while checking to determine if that pixel was denoted as a cell wall. In the case where the pixel is a cell wall, we wrote the radius value into an output memory at the appropriate angle location. Figure 9 shows this architecture.

This architecture operates in a streaming and pipelined fashion. Furthermore it can be parallelized. We used four *Coordinate Conversion* modules in our final implementations; one for each of the four quadrants in the Cartesian plane.

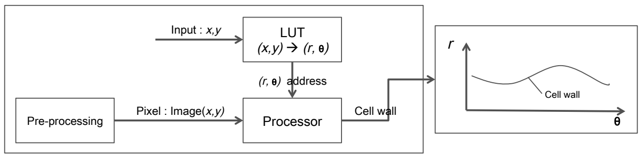


Fig. 9. The *Coordinate Conversion* FPGA implementation.

4.1.4. *HLS Optimizations*

Vivado HLS allows for design optimization using specialized directives. Using C like syntax, we describe the basic architecture. Additional pragmas allow the architecture to be adjusted for hardware execution. Our design primarily uses the *partition*, *pipeline* and *dataflow* directives.

By default, Vivado HLS uses BRAM for each array. The *partition* directive allows arrays to be split across BRAMs and/or implemented entirely as registers. Serialized BRAM memory accesses can limit the parallelism. This directive is applied in almost all modules in our design.

Our image processing algorithm often performs iterative operations across pixels and/or windows in the image. In these cases, pipelining is an effective method to increase the throughput of the design. The *pipeline* directive tells the Vivado HLS tool to create a pipelined version of the indicated code. You can supply the desired initiation interval (II). We typically set $II = 1$, yielding output data every clock cycle.

Lastly, the *dataflow* directive allows for more coarse grain pipelining. We used it to pipeline five modules in our design (see Figure 12(b)).

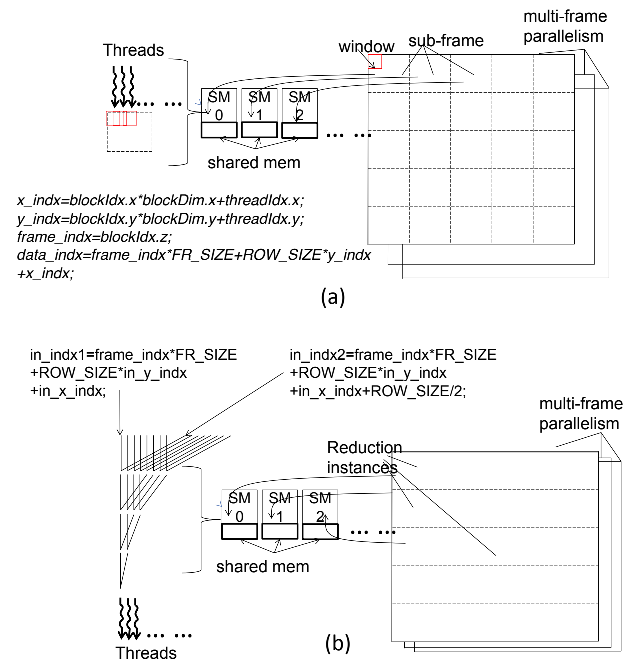
4.2. GPU Implementation

In our GPU implementation, we implemented pixel and frame level parallelism over all the algorithm stages. Since there is no dependency between frames, the GPU can process multiple frames concurrently by simply replicating thread assignment for multiple instances. We developed three sets of CUDA kernels to accelerate the three types of operations corresponding to the FPGA implementation: shifting window, searching, and conversion operations. *Filtering*, *Dilation*, *Erosion*, and *Interpolation* is the first kernel group. *Find Cell*, and *Find Center* are in second group, and subtraction and *Coordinate Conversion* are in the last group. Each kernel is implemented in similar manner.

The first set of kernels accelerate window operations. The hierarchy of the parallelism is demonstrated in Figure 10 (a). We divide each frame into multiple sub-frames, suitably sized for a single thread block. Then we assign each thread to a single window computation within each sub-frame. We exploit the memory locality of the window operations due to two features of the algorithm: the overlapping pixels between neighbor windows and the data dependency from previous operations. Thus, we store the pixels of each sub-frame in the shared memory to avoid unnecessary GPU global memory accesses.

Our second set of kernels use the parallel reduction method for thresholding, finding maxima, and summation. These operations are used when the algorithm searches for the cell location and when locating the center of the cell (Figure 10 (b)). The array lengths of the reduction operations are less than the maximum number of threads per block thread. Therefore, one complete reduction operation can be processed within a single GPU streaming multiprocessor (SM). The intermediate data is stored in shared memory since the reduction operation is conducted within a single SM.

The third set of kernels conducts fully independent operations on each pixel for subtracting two images and converting image coordinates. These kernels are highly parallel. Thus, global memory coalescence becomes the most significant factor for their performance. We ensure that the global memory accesses are coalesced by assigning the appropriate thread block dimensions for the kernels.



```
x_indx=blockIdx.x*blockDim.x+threadIdx.x;
y_indx=blockIdx.y*blockDim.y+threadIdx.y;
frame_indx=blockIdx.z;
data_indx=frame_indx*FR_SIZE+ROW_SIZE*y_indx
+x_indx;
```

```
in_indx1=frame_indx*FR_SIZE
+ROW_SIZE*in_y_indx
+in_x_indx;
```

```
in_indx2=frame_indx*FR_SIZE
+ROW_SIZE*in_y_indx
+in_x_indx+ROW_SIZE/2;
```

Fig. 10. GPU implementation thread arrangements. The data indexing method is described using pseudo code. Part (a) shows the thread arrangements for window operation kernel: each sub-frame is assigned to a streaming multiprocessor (SM) and stored on its shared memory; each window operation is assigned to a thread. Part (b) shows thread arrangements for the reduction kernels: each instance of reduction method is assigned to a SM; the intermediate data is stored on the shared memory.

5. EXPERIMENTAL RESULTS

5.1. Experimental Setup

Our experiments use a 2.4 GHz Intel Quad Core Q6600 workstation for the MATLAB and C results. For the FPGA implementation, we targeted a Xilinx Virtex 6 (XC6VLX240T) using Vivado Synthesis Suite (Version 2012.2). The GPU implementation was tested using a NVIDIA GTX590 GPU with the CUDA 5.0 framework.

5.2. Results and Comparison

Figure 11 shows the latency of the modules in the FPGA implementation. The latency is defined as the number of clock cycles × the clock period in terms of number of clock cycles and absolute time (ms). The first bar provides the baseline results (without directives) and the second shows the results the using *pipelining* and *partitioning* directives in Vivado HLS and performing bit width optimizations on the variables.

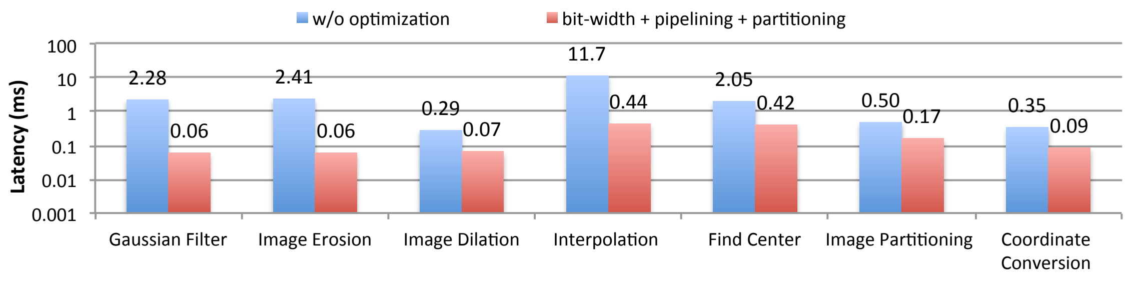


Fig. 11. The performance of each module using different HLS optimizations in terms of latency(ms).

Table 1. Performance of modules in the FPGA design: Latencies in both terms of the number of clock cycles and time (ms) and throughput (FPS).

	Latency (*cycles*)	Latency (*ms*)	Throughput (*FPS*)
Gaussian Filter	6661	0.067	14925
Image Erosion	6661	0.067	14925
Image Dilation	7394	0.074	13513
Interpolation	43890	0.442	2262
Find Center	41753	0.421	2375
Image Partitioning	16386	0.165	6061
Coordinate Conversion	8462	0.085	11764

Table 1 shows the same optimized latency results including the number of clock cycles for each module which factors into the overall latency. Additionally, it provides the throughput in frames per second (FPS). Again this is for each module in the FPGA implementation. The *Interpolation* has largest latency and the lowest throughput and hence is the bottleneck. In a pipelined architecture, this module dictates the overall throughput which is 2,262 frames per second.

Figure 12 compares sequential and pipelined FPGA designs. The total latency is 1.4 ms to detect a cell and extract radius information in one frame. If the system starts to process a new image after finishing one image, its throughput performance will be 714 FPS (see Figure 12(a)). However, we can perform function level pipelining using the *dataflow* pragma. This causes the design to be pipelined across the five major parts of the image analysis as depicted in Figure 12(b). In this design, the total latency is 140,900 cycles and a new frame image can start every 43,890 cycles. That means the latency to process one image is 1.4 ms and the system throughput is 2,262 FPS.

Cell analysis is based upon the deformation as the cell moves through the flow region. In a typical experiment, a cell will appear in approximately 25 consecutive frames in the flow region. The decision on how to sort the cell must occur in under 10 ms in order to be done in real time. The actual calculation that performs the sorting depends on the experiment being performed, but extracting the radius information is by far the most computationally intensive part. For example, the cell sorting could be performed using a threshold based on the maximum deformation of the cell in the channel. Therefore our latency calculation only includes the time to extracting the cell radius information. The total latency to process a single cell across 25 frames takes 0.44 × (25 - 1) + 1.4 = 11.94 ms as shown in Figure 12(c).

The target FPGA was a Virtex 6 (XC6VLX240T). The entire design utilizes 40.07% of the slices (15327 of 37680), 25.84% of the LUTs (38941 of 150720), 6.07 % of the FFs (18303 of 301440), 6.37% of the DSP48Es (49 of 768), and 33.29% of BRAMs (277 of 832).

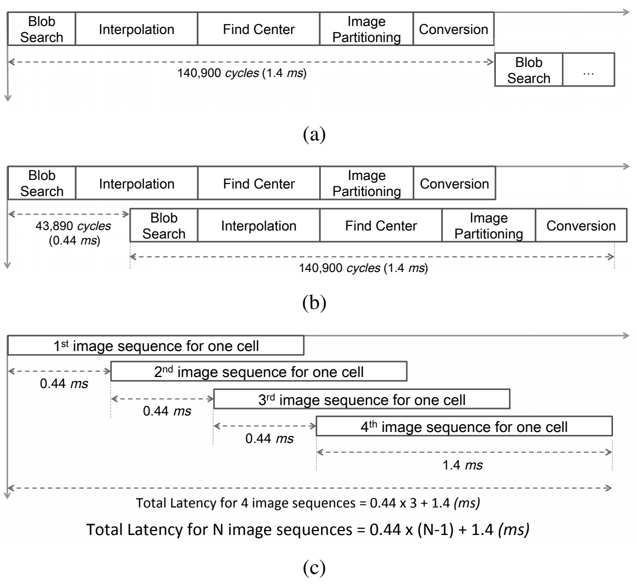
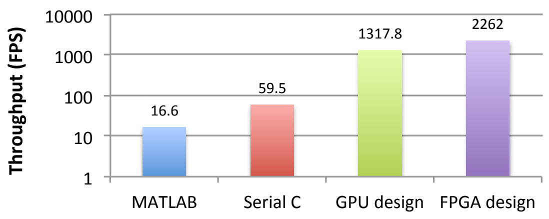
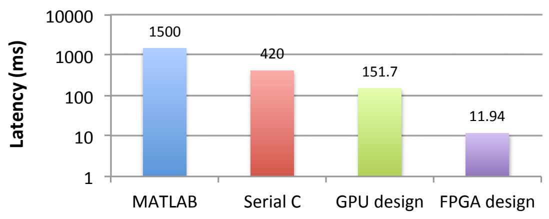


Fig. 12. Sequential and pipelined implementations: (a) the sequential design (latency for one image: 1.4 ms, throughput: 714 FPS), (b) the pipelined design using the *data flow* directive (latency for one image: 1.4 ms, throughput: 2,262 FPS), (c) a method to calculate the total latency required for cell sorting.



(a)



(b)

Fig. 13. A comparison of the performance of the different implementations : MATLAB, Serial C, GPU, and FPGA (a) the throughput (b) the total latency to analyze a series of images for one cell.

Using Vivado HLS shortened our implementation time considerable as compared to handwritten HDL. It is easy to describe the intended architecture using their C like syntax and pragmas for hardware control. This also allows us easily expand our design and add complexity after optimizing and reaching optimal performance in each module. We feel there is great potential to explore further designs with the FPGA and Vivado HLS.

Figure 13(a) compares the MATLAB, Serial C, GPU, and FPGA designs in terms of latency and throughput. Our FPGA design has approximately 38$\times$ more throughput (FPS) and 35$\times$ lower latency than the Serial C version. Our GPU has approximately 22$\times$ and 2.8$\times$ better throughput and latency, respectively. The latency results also show that the FPGA is more suitable than the GPU for this application. Here, the total latency is the duration from the first appearance of a cell to getting the result of the last image for it. The total latency for one cell analysis is 11.94 ms with the FPGA and 151.7 ms with the GPU (see Figure 13(b)). The target latency is 10 ms. and therefore further work is required to meet this requirement.

The GPU latency essentially eliminates the potential to use the GPU as a hardware acceleration platform if we wish to perform real time cell sorting which necessitates a latency around 10 ms. The GPU latency is unlikely to decrease significantly even if with a larger GPU. This is due to the fact that the GPU works through the CPU to transfer data from the camera to the GPU for acceleration. So while additional optimizations may increase the GPU throughput, the latency is largely a function of the transfer time between the CPU and GPU. Therefore, the FPGA is much more attractive option because it can directly connect to the camera, receive the image data at pixel rate, and operate on it in a streaming pixel by pixel manner.

6. CONCLUSION

We have examined a high-throughput biomedical imaging system using two different hardware acceleration architectures: FPGA and GPU. Our experimental results show that both architectures provide considerable performance improvement over a software-only design. And the FPGA design achieves a throughput rate twice as high as the GPU design, 2,262 frames per second(FPS) and 1,317 FPS, respectively. The GPU design also suffers from significantly higher latency, 151.7 ms as compared to 11.9 ms for the FPGA.

7. REFERENCES

[1] D. Gossett, H. Tse, S. Lee, Y. Ying, A. Lindgren, O. Yang, J. Rao, A. Clark, and D. Di Carlo, “Hydrodynamic stretching of single cells for large population mechanical phenotyping,” *Proceedings of the National Academy of Sciences*, vol. 109, no. 20, pp. 7630–7635, 2012.

[2] H. Tse, P. Meng, D. Gossett, A. Irturk, R. Kastner, and D. Di Carlo, “Strategies for implementing hardware-assisted high-throughput cellular image analysis,” *Journal of Laboratory Automation*, 2011.

[3] D. Buschke, J. Squirrell, H. Ansari, M. Smith, C. Rueden, J. Williams, G. Lyons, T. Kamp, K. Eliceiri, and B. Ogle, “Multiphoton flow cytometry to assess intrinsic and extrinsic fluorescence in cellular aggregates: Applications to stem cells,” *Microscopy and Microanalysis*, vol. 17, no. 04, pp. 540–554, 2011.

[4] J. Xu, N. Subramanian, A. Alessio, and S. Hauck, “Impulse c vs. vhdl for accelerating tomographic reconstruction,” in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE, 2010, pp. 171–174.

[5] C. Johnston, K. Gribbon, and D. Bailey, “Implementing image processing algorithms on fpgas.”

[6] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, “Accelerating compute-intensive applications with gpus and fpgas,” in *Application Specific Processors, 2008. SASP 2008. Symposium on*. IEEE, 2008, pp. 101–107.

[7] K. Gribbon and D. Bailey, “A novel approach to real-time bilinear interpolation,” in *Electronic Design, Test and Applications, 2004. DELTA 2004. Second IEEE International Workshop on*. IEEE, 2004, pp. 126–131.

[8] K. K. Zaffini. (2006) Dsps + fpgas provide flow cytometry and cell sorting solution.