UNIVERSITY OF CALIFORNIA, SAN DIEGO

Designing Hardware Accelerated Systems for Imaging Flow Cytometry

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering (Intelligent Systems, Robotics, and Control)

by

Dajung Lee

Committee in charge:

Professor Ryan Kastner, Chair
Professor Truong Nguyen, Co-Chair
Professor Pamela Cosman
Professor Tajana Rosing
Professor Dean Tullsen

2017

The dissertation of Dajung Lee is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____
Co-Chair

____
Chair

University of California, San Diego

2017

DEDICATION

I would like to dedicate this thesis to my family and friends. To my parents who give me the greatest love and believe in my potential. My brother Junsoo who has seen all my struggles next to me and truly supported me. He is a big part of my Ph.D. and my life. To all my friends in my hometown and San Diego. I couldn't name them all, but without their support this long journey would not have been possible. Finally, to Sunghwan, Dr. Bae, he has been my greatest supporter and sent me a strong commitment since 2005.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Ryan Kastner for all his support during my graduate study. He supported me from very early stage of my research until the end and inspired me in research path. I sincerely thank him for all his adviace and guidance while pursuing my research.

I would like to thank my doctoral committee members. Professor Truong Nguyen, Professor Pamela Cosman, Professor Tajana Rosing, and Professor Dean Tullsen. All their comments and feedbacks are essential in my research progress. Especially I thank Professor Truong Nguyen for his support as a co-chair.

I would like to thank all my collaborators and coauthors. Without their helps, I couldn't make this achievement. Dr. Henry Tse and his company gave me a great motivation to keep me on this research, and it was very enjoyable to collaborate with him. Dr. Pingfan Meng and Dr. Janarbek Matai gave me valuable tips and feedbacks over my research project as well as research life. I would also like to acknowledge Nirja Mehta for her extraordinary work and help in my research project. I would like to thank all Kaster Research Group members for their precious comments and feedbacks on my research. All discussions we had are invaluable and significant in my research.

Chapter 4, in full, is a reprint of the material as it appears in International Conference on Field Programmable Logic and Applications (FPL), Lee, Dajung; Meng, Pingfan; Jacobsen, Matthew; Tse, Henry; Carlo, Dino Di; Kastner, Ryan, 2013. There are small changes in format and phrasing as a chapter within this larger paper. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in full, has been submitted for publication of the material as it may appear in Journal of Parallel and Distributed Computing (JPDC), Lee, Dajung; Mehta, Nirja; Shearer, Alexandria; Kastner, Ryan. There are small changes in format and phrasing as a chapter within this larger paper. The dissertation author was the primary

investigator and author of this paper.

Chapter 7, in full, has been submitted for publication of the material as it may appear in International Conference On Computer Aided Design (ICCAD), Lee, Dajung; Althoff, Alric; Richmond, Dustin; Kastner, Ryan, 2017 (accepted). There are small changes in format and phrasing as a chapter within this larger paper. The dissertation author was the primary investigator and author of this paper.

VITA

2010	Bachelor of Science, in Electronic Engineering Sogang University, Seoul, South Korea
2013	Master of Science, in Electrical Engineering (Intelligent Systems, Robotics, and Control) University of California, San Diego
2015	Teaching Assistant, in Electrical and Computer Engineering University of California, San Diego
2016	Teaching Assistant, in Computer Science and Engineering University of California, San Diego
2012-2017	Research Assistant, in Computer Science and Engineering University of California, San Diego
2017	Doctor of Philosophy, in Electrical Engineering (Intelligent Systems, Robotics, and Control) University of California, San Diego

PUBLICATIONS

Dajung Lee, Nirja Mehta, Alexandria Shearer, and Ryan Kastner, “A Hardware Accelerated System for High Throughput Cellular Image Analysis”, submitted in *Journal of Parallel and Distributed Computing (JPDC)* (under revision)

Dajung Lee, Alric Althoff, Dustin Richmond, Ryan Kastner, “A Streaming Clustering Approach Using a Heterogeneous System for Big Data Analysis”, *International Conference On Computer Aided Design (ICCAD)*, November 2017 (accepted).

Dajung Lee, Roger Moussalli, Sameh Asaad and Mudhakar Srivatsa, “Spatial Predicates Evaluation in the Geohash Domain Using Reconfigurable Hardware”, *Field-Programmable Custom Computing Machines, IEEE 24th Annual International Symposium on (FCCM)*, May 2016.

Janarbek Matai, Dajung Lee, Alric Althoff, and Ryan Kastner, “Composable, Parameterizable Templates for High Level Synthesis”, *Design Automation and Test in Europe (DATE)*, March 2016.

Janarbek Matai, Dustin Richmond, Dajung Lee, Zac Blair, Qiongzhi Wu and Ryan Kastner, “Resolve: Generation of High-Performance Sorting Architectures from High-Level Synthesis”, *International Symposium on Field-Programmable Gate Arrays (ISFPGA)*, February 2016

Dajung Lee, Janarbek Matai, Brad Weals, and Ryan Kastner, “High Throughput Channel Tracking for JTRS Wireless Channel Emulation”, *International Conference on Field Programmable Logic and Applications (FPL)*, September 2014.

Dajung Lee, Pingfan Meng, Matthew Jacobsen, Henry Tse, Dino Di Carlo, and Ryan Kastner, “A Hardware Accelerated Approach for Imaging Flow Cytometry”, *International Conference on Field Programmable Logic and Applications (FPL)*, September 2013.

ABSTRACT OF THE DISSERTATION

Designing Hardware Accelerated Systems for Imaging Flow Cytometry

by

Dajung Lee

Doctor of Philosophy in Electrical Engineering (Intelligent Systems, Robotics, and Control)

University of California, San Diego, 2017

Professor Ryan Kastner, Chair
Professor Truong Nguyen, Co-Chair

Creating efficient, accurate approaches to cytometry is an important problem for clinical diagnostics, biological research, and drug discovery. Cytometry identifies cell types or cell status, separates mature cells from immature ones, detects cancerous cells from healthy normal cells, classifies stem cells during differentiation, and screens drugs based upon how they affect cellular architecture.

Imaging flow cytometry is especially promising since this image-based cell analysis system is capable of capturing highly sophisticated contents while achieving high-throughput analysis. Analyzing cellular images quickly and accurately is a non-trivial problem. These images are commonly obtained at the microscopic level and therefore are very sensitive to light, are often plagued by visual noise, and blur easily. Processing these images is highly data-intensive and computationally demanding. Therefore, even state-of-art approaches can achieve either high-throughput or profile the cell contents, but not both. There have consequently been significant demands for a properly designed algorithmic approach, as well as specialized hardware support for it.

This work presents a hardware-accelerated system design for a real-time imaging flow cytometry technique. The main algorithmic approaches in this work are two-folds: 1) morphological feature analysis to describe cellular features and 2) an image segmentation method to classify irregular cell shapes and separate the cellular membrane and nucleus. It first describes a high-throughput and low-latency system design solution for extracting cellular properties from a high frame-rate video. Our system analyzes cell images to understand their mechanical properties, such as shape, size, circularity, or deformability. This work suggests hardware-friendly algorithms and carefully optimized hardware accelerated systems using a reconfigurable hardware, i.e. Field Programmable Gate Arrays (FPGA). Secondly it describes a streaming data clustering method for image segmentation. Data clustering is commonly used for data analysis but is also a demanding process, even in hardware. The segmentation approach in this work achieves a highly streaming and scalable data clustering solution that runs in the highest throughput in an FPGA while handling high-dimensional data. We evaluate this method and conclude that it outperforms other prior state-of-the-art systems. We generalize our streaming data clustering approach for other clustering problems in various data analysis application domains.

Chapter 1

Introduction

Cytometry is a quantitative analysis technique for understanding various cellular properties. These properties include any characteristics that can explain cells in a biological, chemical, or mechanical way: size, shape, cell count, deformation, DNA contents, molecule contents, life span, structure or particular reaction to external stresses. Such analysis can identify cells and give in-depth understanding about them, such as their behaviors or status. This information can be used for fundamental biological research and practical applications. We can diagnose diseases such as cancer or AIDS based on sorted cells derived from this technique, and we can separate stem cells from cancer cells based on analysis results. There are many other applications that use cytometry techniques, such as diagnosing disease, screening for cancer, sorting cells, monitoring immune systems, screening drugs, and developing regenerative medicine. Exploring different cell properties and characterizing them is a fast-growing research problem with much potential, and there always has been high demand for an automated cell analysis system for accurate, fast, and massive analysis. In this thesis, we describe our main contribution: developing a high-throughput cellular image analysis system on reconfigurable hardware for an automated image-based cytometry technique.

Flow cytometry and imaging cytometry are the most conventional methods for cell analysis. Flow cytometry is capable of analyzing a large amount of cell data in very high throughput, but the method makes it difficult to extract complicated features. Imaging cytometry provides sophisticated analytical results but with very limited performance. An imaging flow cytometry method combines the strengths of both of these methods. It is a label-free method with no need for biological or chemical markers on cell samples, and is capable of measuring different cellular parameters while also achieving high throughput analysis.

On the other hand, the current technology for imaging flow cytometry provides very limited solutions for a real-time system, and it still must overcome several challenges before it can beat commercial cytometry devices. First, because it is based on image or video data, its throughput or accuracy is limited by camera performance. There have been documented cases where image data has been collected in an ultra-fast way, but these studies do not include real-time image analysis. Microscopic imaging can present image quality issues, such as low-resolution data, low contrast, or blurring noise. A proper image analysis algorithm is necessary to enhance poor image quality and extract cellular features and characteristics.

Furthermore, a lack of computing power is a big challenge for accurately processing image data. To analyze cellular images for imaging flow cytometry, the processor must handle a large amount of microscopic image data accurately in high throughput. One example comes from our experimental setup: a cell sorting system aims to analyze thousands of cells, which entails processing more than 60,000 frames in a second. The system must also take into account latency constraints inherent to analyzing tasks and classifying cells based on the results. These high-throughput and low-latency requirements are common system constraints of imaging flow cytometry. So a general imaging flow cytometry system analyzes cellular data separately after capturing images or video using a high-throughput camera on an experimental setup.

Practical application of an imaging flow cytometry technique requires in-depth interdisciplinary research in computer science. There are some efforts to analyze these images and extract cellular figures using highly refined state-of-art computing technologies in signal processing, image processing, and computer vision. However, more accurate feature analysis results in higher algorithmic complexity, which hinders a real-time image analysis.

To overcome these performance constraints and achieve accurate cell analysis, algorithm development and hardware system design must be accommodated. In general, even a state-of-art image analysis algorithm is unable to provide faster than a few milliseconds latency for one frame analysis. Therefore, more robust hardware would increase the computing power of the system.

A GPU and FPGA are common candidates for hardware acceleration techniques. GPU gives massive data parallelism based on SIMD (single instruction multi data) architecture and is able to achieve high throughput by processing input and generating output within a certain time. However, this device uses DDR memory to have necessary data, which causes high latency. FPGA is good at pipelined parallelism and is very friendly for streaming process. It has very limited size of on-chip memories, but has high memory bandwidth within a chip. A carefully optimized hardware architecture on the logic level will achieve a high level of pipelining parallelism and will minimize data access, making it capable of high throughput and low latency constraints simultaneously.

For this research, we propose a novel hardware-accelerated system for a real-time imaging flow cytometry to achieve high accuracy and meet strict performance constraints at the same time. More specifically, our target system captures cell images with a very high-performance camera, processing thousands of cells per second and taking a few milliseconds for a single cell analysis. Despite the fact that these images have low contrast and are noisy (and therefore blurry) due to the fast movement of the cells, our system processes these noisy, low-contrast, blurry images with very high throughput and minimal latency. We are able to achieve this with our image analysis system, which computes cellular mechanical properties using brightfield imaging on a microfluidic device. The system images cells using a high-speed optical image sensor, analyzes the resulting video streams, extracts features from the images, and classifies cells based on those observed features.

Our research then focuses on extracting mechanical properties from these cell images. These properties includes morphology and inner cell structures. To achieve these goals, we carefully develop hardware-friendly algorithms and examine different computing platforms. We mainly focus on developing an accelerated system on an CPU-FPGA system of this algorithms.

The remainder of the thesis is organized as follows: In Chapter 2, we explain the background of this research and introduce a motivative system. We also provide an overview of cytometry technologies in this chapter and present related works. In Chapter 3, we introduce the background of hardware acceleration using FPGAs and describe several hardware optimization methods we used in our system. In Chapter 4, we discuss a basic hardware-accelerated approach using different platforms. We also introduce an initial flow of our cellular analysis algorithm and compare our preliminary FPGA-based design with a GPU-based accelerated system. In Chapter 5, we describe an advanced accelerated design on an FPGA system. We show an in-depth image analysis algorithm to extract morphological features and explain its optimized hardware architecture for a real-time performance. In Chapter 6, we extend our cell analysis technique to more complicated features. We explore different image segmentation methods and compare how they would perform in order to achieve efficient hardware implementation. In Chapter 7, we propose an efficient streaming clustering algorithm for image segmentation on an FPGA-CPU heterogeneous system. Finally, in Chapter 8, we summarize our contributions and conclude this thesis.

Chapter 2

Imaging Flow Cytometry Research

In this chapter, we discuss the fundamental background of computational cell analysis research and the motivation for our work in detail. We describe a target imaging flow cytometry system and clarify a system goal in this research. We review other related works and discuss different approaches suggested in various domains in Section 2.3.

2.1 Cytometry Research

Cytometry is a quantitative analysis technique that measures the various characteristics of cells. Cytometry research accesses biological, physical, or chemical characteristics of cells to understand target cells, including cell size, cell count, cell morphology, shape or structure, cellular lifespan, DNA contents, a certain reaction to a particular chemical, or the existence of specific proteins, etc. For example, it is used to count blood cells in common blood tests used in medical diagnostics. Understanding cells has huge practical advantages, as well as implications for future research. Cytometry technology is also used in a variety of cell biology and medical research, such as cancer and AIDS research, as well as in designing regenerative medicines.

Given that cell analysis is a fundamental problem in biological research, cytometry

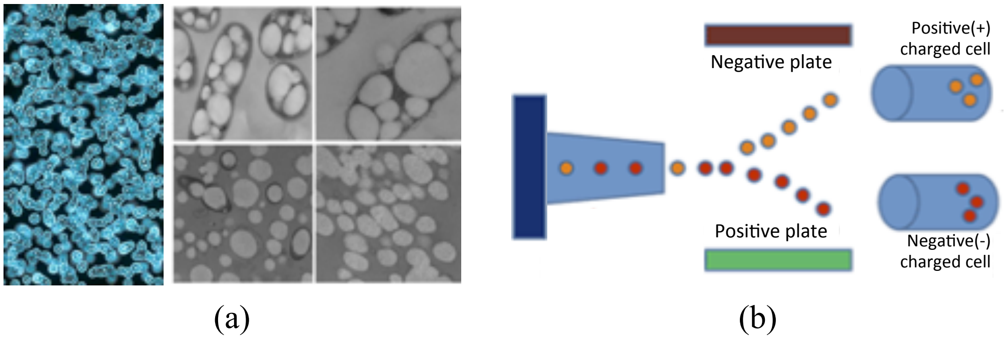


Figure 2.1: Examples of cytometry researches (a) Imaging cytometry assesses cellular properties based on images, which enables a sophisticated content analysis. (b) Flow cytometry is able to analyze cell in high throughput manner using a specialized device.

technology has an extensive history. The two most common methods for cell analysis are flow cytometry and imaging cytometry, while more recent imaging flow cytometry techniques combine the advantages of both approaches. In this section, we review these different cytometry approaches.

2.1.1 Imaging cytometry

Imaging cytometry is the oldest and most fundamental method for analyzing cells. It is based on visible properties of cells (see in Figure 2.1(a)) and observes cells using a microscope or optical sensor devices. The method analyzes cells statically and as digital camera technology introduced in this research, a high resolution microscopic sensor enables sophisticated high-content screening capable of separating cell features. This method, however, cannot be performed in a high throughput manner. Moreover, imaging cells at the microscopic level commonly requires staining them with a fluorochrome, which binds to a structure within the cell [32, 63]. This labeling process highlights particular molecules or cellular structures. For example, it can separate out individual cell features (like the cell membrane or nuclei) and determine interactions between multiple cells [29]. Accurately extracting cell parameters demands significant effort, making it likewise difficult to perform high throughput analysis [21, 58].

2.1.2 Flow cytometry

Flow cytometry is the most commonly used method for high-throughput and low-latency systems. It assesses cells using a specialized device aligned with target properties, making it useful for a broad variety of practical applications. Flow cytometry can use a laser [57, 56], an optical device [45, 31], or an electrical impedance device [19, 25] to extract course features from cells suspended in a fluid. It tags cells with a biomarker – fluorochromes, for example, which activate when targeted with a particular wavelength of light – and then uses a specialized device to detect this marker. This detector does not require a complex processing or a huge computational load to analyze, so it is capable of massive cell analysis. However, feature complexity focusing is limited to a particular feature or a special cellular functionality, and the detector requires a specialized setup that requires cells be labeled prior to screening (see Figure 2.1(b)).

2.1.3 Imaging flow cytometry

Imaging flow cytometry combines the strengths of flow cytometry and imaging cytometry [15, 17, 9]. A modern high-speed camera sensor is able to take complicated feature images of cells in high throughput and can dynamically see cell features with high sensitivity in high throughput. For example, the ImageStream by Amnis [2] is a commercial imaging flow cytometer capable of processing 5,000 cells per second. It produces 12 images – 10 fluorescent markers in addition to darkfield and light-field images. The fluorescent images provide higher contrast but require a pre-processing step to add the fluorochromes. On the other hand, the lightfield and darkfield images have no pre-processing requirement, but have reduced image clarity.

This new technology has clear benefits and high potential to be a state- of-art cytometry technique. It allows label-free cellular analysis for many different types of research and can measure different complicated cellular parameters from a single dataset in high throughput. However, several limitations prevent the technology from achieving real-time cell analysis in any practical sense . First, it is limited by camera performance and its imaging quality. Image quality is commonly low-resolution and low-contrast with considerable noise. Second, it requires a proper image analysis algorithm to accurately and efficiently extract target cellular information from this noisy image. Last, it requires an accelerated system in embedded hardware to build an automated imaging flow cytometer. Because of the massive amount of data and the aforementioned performance constraints, it is hard to process image analysis and achieve real-time performance without strong support at the hardware level, so thus far it is commonly performed offline.

This research, therefore, is highly interdisciplinary and requires further study in a variety of areas, from fundamental understanding of cell biology to algorithmic insight for cell image analysis and system level design intuition. In this thesis, we explore highly refined image processing, computer vision, and machine learning approaches to designing a cellular analysis algorithm, and high-performance computing and hardware acceleration approaches for building a real-time imaging flow cytometry system.

2.2 System Goal

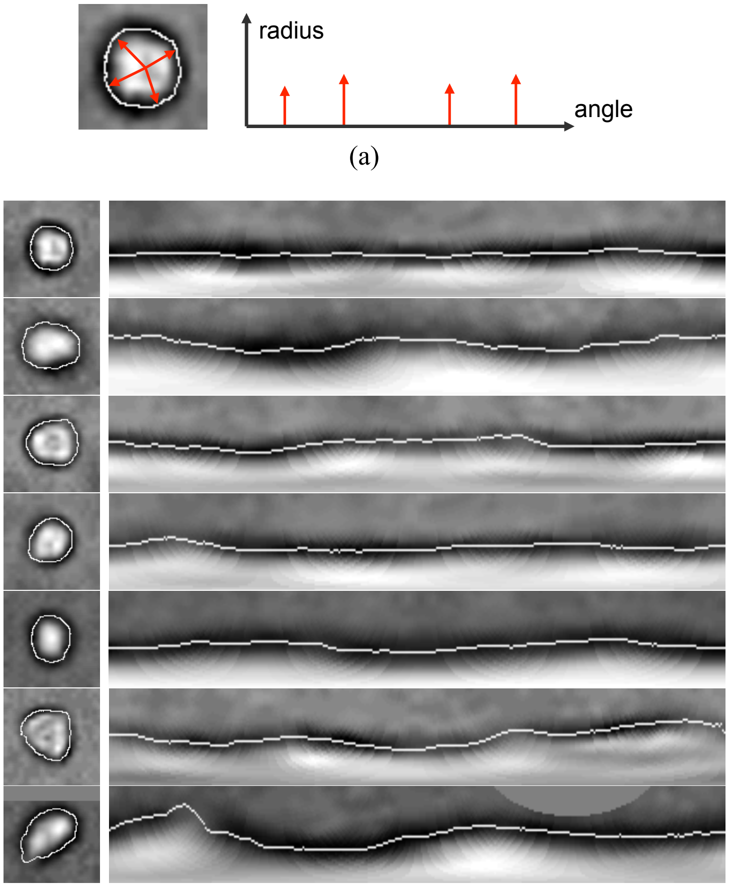
In this section, we present the main cellular properties we want to observe in our cell analysis system, describe our experimental setup for extracting the target features in detail and discuss the performance constraints inherent to achieving a real-time cellular image analysis system.

2.2.1 Target cellular properties

Our system targets a particular type of imaging flow cytometer that analyzes cellular mechanical properties based on its morphological feature by way of bright-field images. Cell morphology is a common and useful biomarker for clinical research of several diseases. This morphology feature can include size, shape, texture, and nucleus-to-cytoplasm ratios. For example, during a stem cell maturation process, its morphological shape or texture can indicate its fully matured cell type, and HIV infected T-cells can be identified by using their morphological change [54]. In our cytometry system, the main idea is to generate a force on a cell in a flowing fluid, measure its morphological feature in depth, and determine its physical response. We can analyze the high speed images to determine mechanical properties based upon the cell's shape, size, circularity, deformability, or its inner structure. And then we can use those features to classify the cell.

Cell morphology

To extract mechanical properties of cells – such as size, circularity, and deformability – we first focus on observing a cellular morphological feature within a frame, i.e. its external shape. If we measure the cellular structure accurately, we can easily estimate other mechanical cellular properties based on these measurements. Figure 2.2 presents examples of cellular morphological shape analysis. Figure 2.2 (a) shows an angle versus radius plot. It converts a regular cartesian coordinate cell shape into a polar coordinate plot based on a center of cell. It will stretch and unroll a round cell feature and make a flat plot. It traces cellular wall and measures distance between a center point and it in every angle between 0 through 360 degree. Figure 2.2 (b) shows cellular wall tracing results, while a cell changes its shape in a experiment. A cell on the top is round and presents a flat radius plot, but it changes the shape in the bottom image and shows a different plot.



(a)

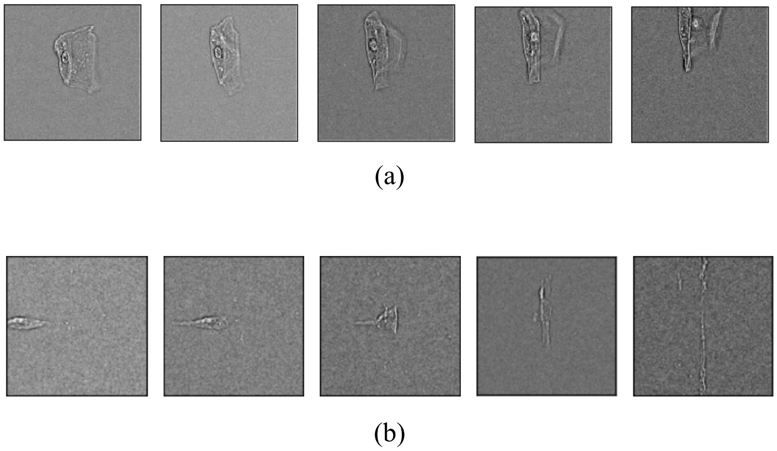
(b)

Figure 2.2: Examples of cellular morphological shape analysis. An image analysis algorithm measures cellular radius in every angle and extract the cellular morphological feature that can describe cell shape in accurate.

For example, we can determine the deformability of a cell by analyzing its image. Different cells will deform in different ways. A pluripotent stem cell deforms more than its differentiated progeny; pleural fluid with metastatic cells will deform more than fluid with normal cells [28]; cells susceptible to tumor cell invasion have a changing mechanical behavior [40]; cancerous cells with the highest invasive potential are stiffer than those with lower migrations [61]; and older cells deform differently than younger ones [67]. More generally, recent research states that cells mechanical properties "play important roles in the regulation of various biological activities at the molecular and cellular level" [72]. Thus, a cellular image analysis system for microfluidic deformability cytometry provides an attractive approach for high throughput cell screening and sorting. In Chapter 4 and Chapter 5, we focus on observing cell morphological shape analysis.

Cell structure

Here we extend our scope to describe more subtle morphological features, such as specific outlier cell shapes and inner cell areas, in order to see more detailed cellular structure. Our approach mainly focuses on morphological shape to obtain an accurate measurement of a cell shape, but this is limited to cells that are round in shape. However, other types of cells are extremely deformable or could potentially explode on our device junction (see in Figure 2.3). To generalize our method for these cells, we explore image segmentation methods and describe our hardware implementation for it in Chapter 6 and Chapter 7.



(a)

(b)

Figure 2.3: Examples of cells with different structure. (a) Different types of cells may have irregular morphological shape or have a separable nucleus from membrane. (b) A cell may explode within a device by a force of flowing microfluid depending on its state.

2.2.2 Experimental setup

This uses a microfluidic channel to deliver a cell into the center of a stretching extensional flow, which generates a uniform stress on the cell, thereby causing a deformation. The cells flow quickly through the microfluidic channel, enabling high-throughput processing. By imaging the cell in the extensional flow with a high-speed image sensor, we can observe the deformation of large population of cells with a high throughput.

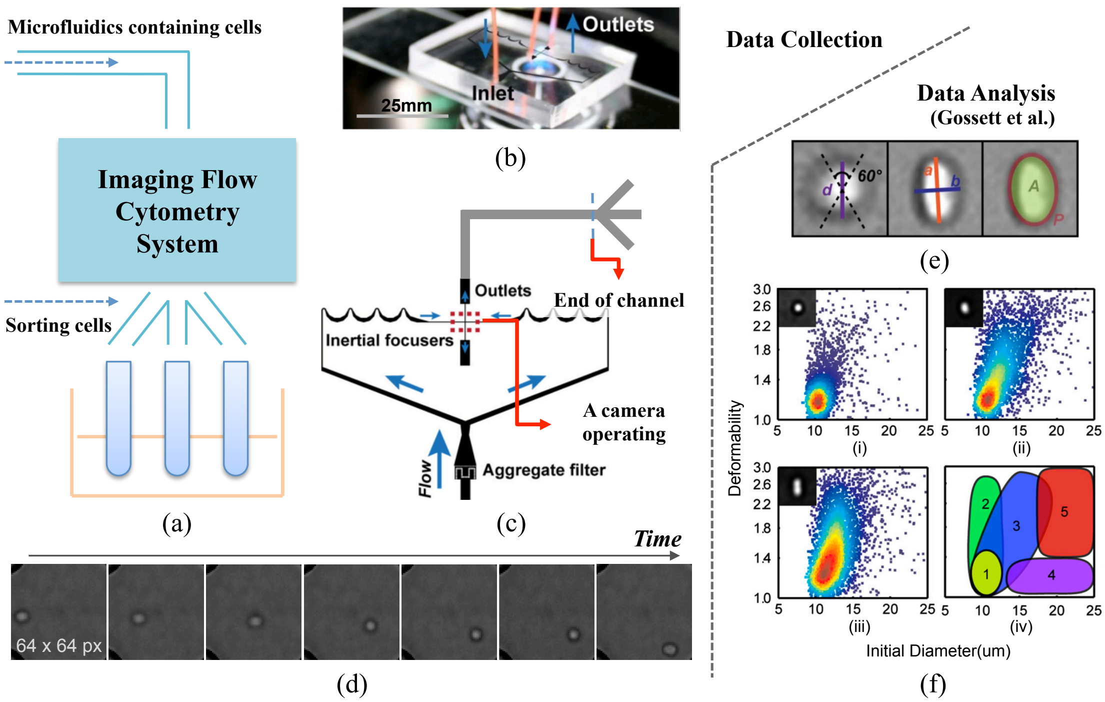


Figure 2.4: (a) An imaging flow cytometry system can be used to sort cells by imaging micro-fluid having cells in a device. It consists of two big processes, data collection and data analysis. Data collection: (b) this system has a microfluidic device where the fluid flows (c) the device is designed to generate uniform pressure to stretch cells. (d) raw images for one cell event. A cell can appear on 15 frames at most from entering to exiting a field of view. Data analysis [28]: (e) it analyzes cellular morphological features, such as initial diameter, circularity, or size. (f) examples of cellular mechanical property analysis, (i),(ii) and (iii) density scatter plots of the size and deformability of different cells. (iv) different patterns for different cells in size-deformability map.

Figure 2.4 shows the target system, which is designed to produce cell stretching in an extensional microfluidic channel. The target system uses a high-speed camera to observe cellular deformation. It applies uniform hydrodynamic force to a single cell on the channel; meanwhile the fast-flowing fluid enters the field of view of the camera. High-speed microscopy focuses on the center point, thereby imaging the cells movement and its deformation. For example, Figure 2.4 (d) shows a sequence of events for a single cell, from entering the field of view to exiting into an outlet for sorting. The resolution of this microscopic image is very tiny less than 128×128, and one cell stays in this view only few microseconds or few frames. Because of the high speed of fluid in the channel, this hydrodynamic approach is able to assess a large number of cells efficiently. This technique has the potential to process up to 20,000 cells per second. However, it is limited by the critical bottleneck involved in handling the generated image data.

Our target performance is to analyze 2,000 cells per second while assuming (1) any frame has no more than one cell; (2) one cell event appears in 7-15 consecutive frames; and (3) only 50% of frames have valid cell features. That means the system must process 28,000-60,000 frames in one second. Also, for the practical usage of cell sorting, it should analyze one cell within a predetermined latency, which is dictated by the time it takes the cell to arrive at the sorting point or the end of the extensional channel.

2.3 Related Works

In this section, we review other literature pertaining to high-performance hardware-accelerated cytometry research. High-throughput cell analysis systems are mostly based on an FPGA, DSP or embedded-sensor system. As we discussed, the most conventional cytometry research for high throughput analysis is flow cytometry, which reads a particular reaction signal that varies depending on the experimental setup and cellular properties. This specialized hardware has been commonly employed in various signal-processing research and provides an optimized system to meet high throughput and low latency constraints. Thus, flow cytometers for high-throughput cell analysis are, in general, implemented on an embedded system to maximize their performance.

There are several imaging flow cytometry approaches, but mostly they are limited to a particular type of image input that still requires the researcher to biotag or attach fluorescence on a cell or risk limited performance. [15, 26, 27] Our approach is based on a bright-field image taken by a high-speed phantom camera, which requires no tags on a cell. We measure cellular morphological features based on this image input enabled by a high-performance system using an FPGA.

Buschke et al. [15] present a cell analysis system for image cytometry using a DSP and FPGA. Their system attempts to detect the cell and automatically process the image in the same way we do. However, our input data are bright field images, which require more complex processing than their fluorescence images. Moreover, our requirements are more challenging in terms of throughput and latency compared to their system, which operates at 2.33 frames per second.

Goda et al. [26, 27] developed a heterogenous hardware accelerated (FPGA and CPU) image cytometer for cancer cell detection. For this application, the FPGA is used for data capture and performs only simple low-level processing, which amounts to coarse size-based classification. The CPU performs the bulk of the analysis on the filtered images. Our system requires significantly more advanced morphological feature detection to be performed in real-time on the FPGA.

Some of acceleration approaches use a GPU accelerator. Tse et al. [62] present a GPU-based approach to analyzing cellular morphological features. It parallelizes the morphological analysis by mapping a pair of images at two different coordinates and bottomhat filtering them for contrast enhancement. This approach shows a great improvement in performance, but only partially accelerates the entire image analysis pipeline. A GPU basically parallelizes processing operations with a large number of processing elements, but it does use DDR memory and has to keep reading data from it. This causes inevitably high latency and is not able to meet our latency constraints. Our basic hardware result shows that an FPGA approach presents 1.9 times faster throughput with 107 times better latency than a GPU approach in Chapter 4.

Using hardware acceleration for image processing is well known in many other domains. However, the target throughput is generally less demanding (fewer than 1,000 frames per second) and is focused on processing the larger resolution images [37, 30]. Greisen et al. present a video-processing pipeline for high-definition stereo video in [30]. It utilizes a FPGA-GPU-CPU system for high-speed stereo vision and processes video streams up to the resolution 1920×1080 pixels at 30 frames per second.

There are several works that accelerate a medical imaging system on an FPGA [66, 20]. Coric et al. [20] present a hardware-accelerated parallel-beam backprojection algorithm used in computerized tomography (CT). Xu et al. [66] developed a medical imaging system for a CT filtered backprojection algorithm. They compared and investigated different hardware designs using C, Impulse C and VHDL. Their work is limited in that they show their manual design has better performance than Impulse C. Our work targets a different imaging system and achieves the high performance requirements.

We are aware of several other image processing systems that run at higher frame rates (thousands of frames per second) [38, 35, 27]. Kagami et al. show a networked vision system of transferring visual features using ethernet at 1,000 fps in [38]. It handles a 64×64 pixels image on an FPGA attached to a CMOS vision chip, but it does not analyze the image itself and only performs on preprocessed vision features, transferring them through a network. IDP Express is a high-speed vision system that uses an FPGA to record 512×512 pixels images and operates at 2,000 frames per second [35]. Compared to these works, our target system has more challenging requirements, as it needs to perform more complex image analysis to extract the cellular morphology.

Chapter 3

Hardware Acceleration on FPGAs

3.1 Introduction

With the insistent growth of information technology and computer industry in the domains of IoT (Internet of Things), autonomous driving, wearable healthcare, data center, and 5G networks, the high performance systems have witnessed notable popularity and demand in the global market. There are several approaches that aim at improving their performance at different system levels. Recently, hardware accelerated approach has received attention from academia and industry as we are facing limits in our traditional approaches in high level. While a software-based approach on a conventional general-purpose computer is hard to achieve the demanding system goals, a hardware acceleration approach provides an application specific system solution with a customized architecture using hardware, such as FPGA, ASIC, DSP, or GPU. However, the difficulty level increases with the inclusion of several design constraints in the designing process, which are related to acquiring high performance for a real-time system, small resources for the cost, less power for energy efficiency, and small size form factor for mobile devices.

An FPGA-based approach is considered to be an efficient method in meeting the aforementioned constraints. It provides a reconfigurable hardware solution design for a specific application. The architecture of an FPGA-based approach is highly customized for certain operations with different level of parallelism, so as to achieve higher throughput and lower latency when compared to software approaches. It is also renowned for using smaller amount of power in a single chipset, in contrast to larger full systems. It is further reprogrammable for different applications in a field at an inexpensive price. Due to these advantages of FPGA design, it has been adopted in various applications ranging from digital signal processing [41], communications [50], biomedical imaging [42], computer vision [48], data analytics [43, 51], to data center [55] and deep learning [13], etc.

The our research is centered around an FPGA-based acceleration, where the design constraints are focused on high performance, throughput and latency. As our system algorithm requires complex computations to handle large amount of image data for fast cell analysis, it is impossible to achieve the target performance in a CPU-based system. Also, an end-to-end system on a single chip can be efficiently extended for a small device such as a camera with an embedded FPGA. Therefore in this research, we achieve high performance cell analysis system with an FPGA. However, there are still several difficulties on an FPGA-based acceleration, where design complexity is one of the most tedious challenges among the others. Achieving an optimized architecture requires skillful expertise, experience, effort, and time. For easing the complexity, recent investigations have been conducted, where hardware architecture is synthesized by using a High Level Synthesis tool. In this chapter, Section 3.2 introduces a High Level Synthesis; Section 3.3 explores several fundamental optimization methods to understand our achievement; and Section 3.4 highlights an optimization example by using the synthesis tool.

3.2 FPGA design using High Level Synthesis

FPGA-based hardware acceleration has several benefits in a high performance system design. It can improve the system computing power and achieve extensive performance gains over a CPU-based system. A designer can finely customize hardware architecture on an FPGA with high flexibility; and further, its re-programmability enables the modification on the architecture without hampering with the system.

Apart from these advantages, when looking into the shortcomings of an FPGA design, it is essential to note its high complexity. The designer is fully responsible for exploring a design space, verifying the functionality of cores, and optimizing the hardware logics considering task scheduling at low level or clock timing issues. Such design sequence is non-trivial work, for which, several commercial tools can be employed at each stage to fulfill the above described design requirements. Thus, in-depth knowledge of hardware and considerable experiences pertaining to system are some of the common requisites essential for developing and acquiring an optimal quality of design.

In the recent past, several efforts have been put into minimizing the design complexity and developing an FPGA that can be easily implemented in broad application areas. The High Level Synthesis (HLS) tool is one of such efforts, which provides an API for an FPGA design by utilizing a higher level programming language, such as C/C++, Scala, Haskell or MATLAB. There are several well-known HLS tools; Vivado HLS [7], OpenCL [4], Catapult High Level Synthesis [1], Synopsys Synphony C Compiler [6], LegUp [16], and Simulink [3].

Figure 3.1 presents the design flow using a high-level synthesis tool, Xilinx Vivado HLS, which we mainly use in our research. The HLS tool provides a directive-based API for hardware designing and FPGA optimization. It rises up its abstraction level and allows the users to build a hardware using C/C++/SystemC languages. It takes

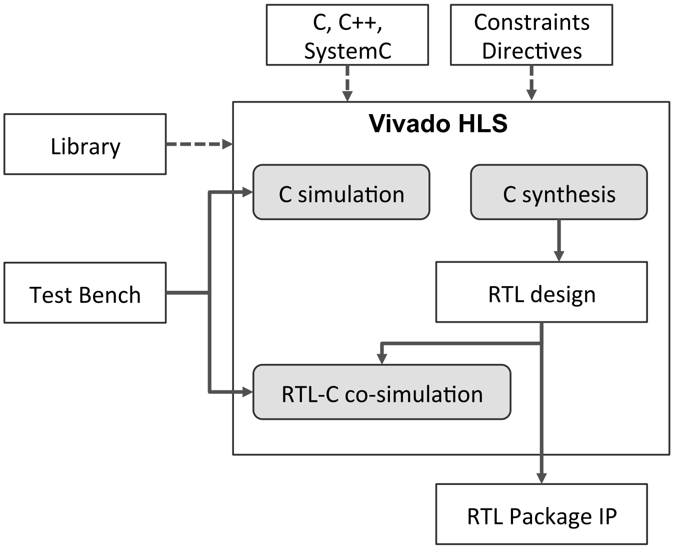


Figure 3.1: Vivado HLS design flow. Vivado HLS is a directive based hardware design API. It takes C/C++/SystemC codes as input, synthesizes them as directives defines, and generates an RTL core.

a C-like code as an input for generating an output of RTL design. As described in a user-defined directive or a constraints file, it synthesizes the input code. The RTL IP core generated by the tool can then be integrated with the other cores or merged into a larger system. Other HLS tools accommodate a similar design flow in a hardware design process.

However, in spite of the higher abstraction of HLS tools, a design process still requires someone with adequate application domain knowledge with hardware architecture background. It does not only require novice level hardware designers or software programmers, but also experienced hardware designers to start and execute a system prototype while exploring the design space in a short time [52]. In the rest of this chapter, we will introduce some fundamental background of FPGA design using the High Level Synthesis tool. Though we mainly used Xilinx Vivado HLS, comprehending technical architectural background of FPGA design could be beneficial for other tools.

3.3 High Level Sytnehsis Optimizations

In this section, we will introduce important background of hardware architecture and present several optimization methods using HLS tool, such as different parallelisms or memory configurations.

3.3.1 Performance Estimation

We use two metrics to measure performance of a system, throughput and latency. Figure 3.2 presents basic definitions of them.

Latency Latency is the total amount of time required to perform an operation; from initiating to finishing it. It is measured in units of time, such as seconds, minutes, or hours. In hardware design, the number of clock cycles can be used to estimate the total latency, which will be converted to a timing unit using a clock frequency of the operation.

Throughput Throughput is the total number of operations performed within a fixed unit time. It is measured in units of format that can estimate amount of operations performed, such as output data generated, iterations, sample data processed, etc. For example, we use the number of analyzed cells in a second (cells/sec) in our research, or it can be the number of frames in a second (FPS) in terms of cell image frames.

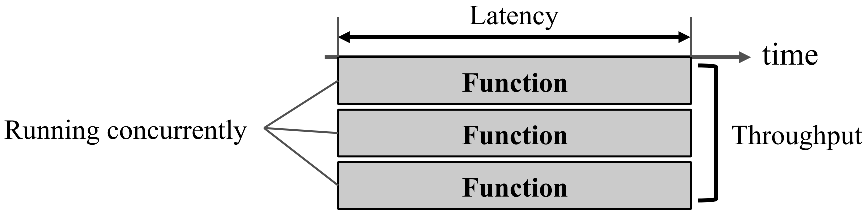
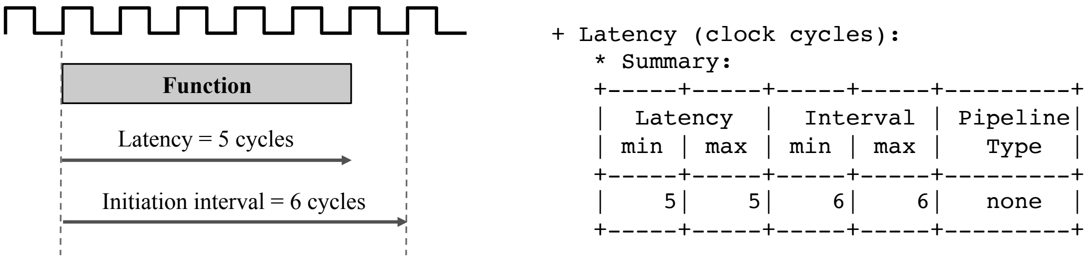


Figure 3.2: Metrics for performance estimation, throughput and latency. Latency is a running time measured from the beginning of a function until the end of it. Throughput measures the total number of functions or operations performed within a unit time.

In a single core-based general purpose system, operational functions run on a processing core in a sequential manner. So the throughput is an inverse of the total running time, i.e. latency. A shorter latency means a higher throughput in a single core system. However, in a parallelized system, the latency and throughput are not inversely correlated. For example, a GPU can add up more processing unit cores for parallel operations as long as cores are needed and available and can increase the throughput, but that does not decrease the latency of the operation necessarily.

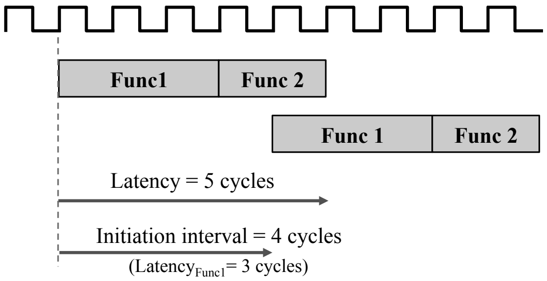
In Vivado HLS tool, a design synthesis report gives an estimated clock cycles for a latency of a function (see in Figure 3.3). We can calculate latency and throughput from this report. The latency is the number of clock cycles to run a function, and initiation interval (II) is the minimum latency in clock cycles to launch the next operation sequence. In an example in Figure 3.3, a total latency for a given function is 5 cycles, and it needs 6 cycles to accept data for next process and start it.



```
+ Latency (clock cycles):
   * Summary:
   +-----+-----+-----+-----+---------+
   |  Latency  |  Interval | Pipeline|
   | min | max | min | max |   Type  |
   +-----+-----+-----+-----+---------+
   |    5|    5|    6|    6|   none  |
   +-----+-----+-----+-----+---------+
```

Figure 3.3: An example of synthesis report from Vivado HLS. It gives latency and initiation interval measurements in terms of clock cycles. Initiation interval is used to calculate throughput.

Initiation interval(II) is used to calculate throughput. In a pipelined operation as an example in Figure 3.4, it starts a new sequence every 4 cycles. A smaller interval means that it can launch a new operation sooner and process more data or more functional operations within the same time window. So a smaller II means a higher throughput. The optimization process using Vivado HLS mostly focuses on minimizing the number of clock cycles for initiation interval(II) in different granularity levels.



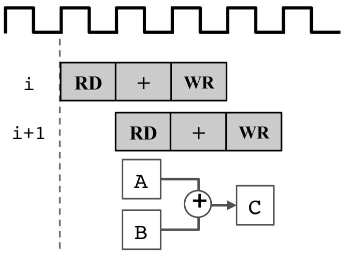
```
+ Latency (clock cycles):
   * Summary:
   +-----+-----+-----+-----+---------+
   |  Latency  |  Interval | Pipeline|
   | min | max | min | max |   Type  |
   +-----+-----+-----+-----+---------+
   |    5|    5|    4|    4| dataflow|
   +-----+-----+-----+-----+---------+
```

Figure 3.4: A latency and initiation interval in a pipelined operation. Initiation interval lowers after pipelining and give higher throughput. The throughput for a entire sequence balances for a bottleneck module, Func1 in this example.

3.3.2 Parallelism

One of big advantages of hardware design is the high flexibility to cosutomize every cores in logic level. Hardware designer can finely parallelize their design on an FPGA and improve the performance. Pipelining is the most important technique that achieves parallelism in instruction level or operation level. It can be done in a fine level operation or be a higher functional level pipeline. An HLS tool provides a pragma or directive for them.

Loop Pipelining



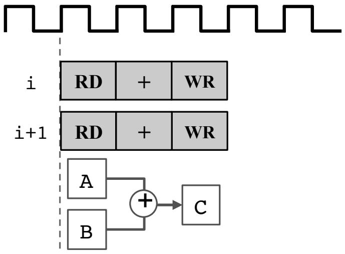
```
void foo(BYTE A[N], BYTE B[N], BYTE C[N]){

    for ( i=0 ; i < N ; i++ ){
    #pragma HLS PIPELINE II=1
        C[i] = A[i] + B[i];
    }
}
```

Figure 3.5: An example of usage for `#pragma HLS pipeline`. User can indicate the target interval as `II=1`. If there is no data dependency between iterations, it pipelines the loop iteration code lines automatically. In this example, it can process one input or one operation as input is ready in every cycles (`II=1`)

Loop pipelining is to achieve pipelining parallelism in an instruction level for an iterative operation in a loop. Figure 3.5 presents an example. HLS automatically converts this for loop code block into a functional module accepting two input and generating a single output. A baseline code with no pragma processes the operation sequentially and wait for the function completes the first iteration. With a pipeline pragma, it will launch the next iteration as soon as the module is available for accepting a new input.

We can define a desired initiation interval (II), the minimum latency to launch a next process, with the pipelining pragma. The arguments following with the pragma tell the HLS tool what II we want to achieve, i.e., the target throughput. In our example above, a target operation in a loop is processed as soon as a new data is ready (RD), which is one clock cycle, interval one. The pragma is define with an argument, `II=1`. A loop operation in Vivado HLS is a basic programming function used in most of applications. An iterative pixel-wised operation over an image is one of examples in computer vision.

Unroll



```
void foo(BYTE A[N], BYTE B[N], BYTE C[N]){

    for ( i=0 ; i < N ; i++ ){
    #pragma HLS UNROLL
        C[i] = A[i] + B[i];
    }
}
```

Figure 3.6: An example of usage for `#pragma HLS UNROLL II=1`. It parallelizes independent operations considering input and output data bandwidth and maximizes the throughput.

Figure 3.6 presents an example of loop unrolling. It provides different parallelism within a functional loop. If there is no data dependency and each iterative operation is independent from others, the example summation operation can be processed simultaneously. Unroll pragma also accepts the user defined unrolling factor, which is presented II. In our example in Figure3.6, II is two. It can be partiall unrolling or completely partitioned unrolling. Unrolling makes multiple copies of the operational module and improves throughput, but depends on the data dependency and consumes many resources.

Dataflow

Dataflow pragma is for a task level pipeline. It is usually used in a top function that calls multiple submodules. It pipelines these submodules and schedules them to start their operation as soon as the module and data are available. Figure 3.4 in Section 3.3.1 presents the task level pipelining. The overal latency to finish two functions is 5 cycles, but when they are pipelined, the next function can be launched in 4 cycles before the first fucntion finishes its job.

This high level pipeline will reduce initiation interval in an overall sequence, which means higher throughput. The performance is determined by a functional module that has the most latency, which is a bottleneck; in our example, it is Func1. Other modules balance out their performance with the bottleneck module. So for higher performance, splitting submodules into finer functions may give a better performance result. However, it consumes more memory to hold data between submodules as a tradeoff.

3.3.3 Memory Configuration

Loop pipelining and unrolling provides directive based hardware optimizations for an instruction level and a task/data level parallelism. In most of practical cases to maximize the system performance, they should map memory configuration directives as well. Even if the operational performance is highly optimized, it is hard to achieve the most performance with no data fed enough to the core.

An FPGA has small size on-chip memories, block-RAMs(BRAMs), which can be configured in different ways depending on a desired architecture. It closely locates with computation logics, so it can be quickly accessed. It takes only two cycles at most to read data point from a BRAM. However, it is very tiny compared with DDR memory. Especially when entire modules are running in functionally pipelined way, it consumes a lot of FIFOs between modules. Depending on how to optimize the memory resource, we can have higher memory bandwidth and achieve better throughput as well. So the data layout should be carefully considered to optimize the architecture. BRAM resources in an FPGA can be easily reconfigured in different ways. Two memory configuration methods we mainly used are *partitioning* and *reshaping*.

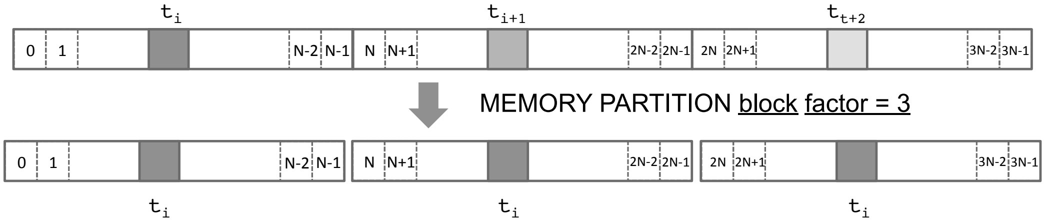


Figure 3.7: An example of memory partitioning. It splits a large array into multiple smaller memories. It improves memory bandwidth, and a core connected can take multiple data simultaneously, one each from a separate memory.

Partitioning Memory partitioning splits a single array data into multiple arrays and allocates different memory resources. Each memory is accessible independently. So we can have higher memory bandwidth. Figure 3.7 presents an example of memory partitioning. Before partitioning the memory, it will take several clock cycles to read multiple data from a single memory. We can read three data component separately one by one in Figure 3.7. However after partitioning this array into three different memories, we can retrieve these data simultaneously.

This memory partitioning is useful when it needs a parallel operation with multiple data. For example, sliding window operation in image processing needs to read a convolution kernel coefficient. If memory for coefficient are partitioned, then we can read these data concurrently and process them in parallel.

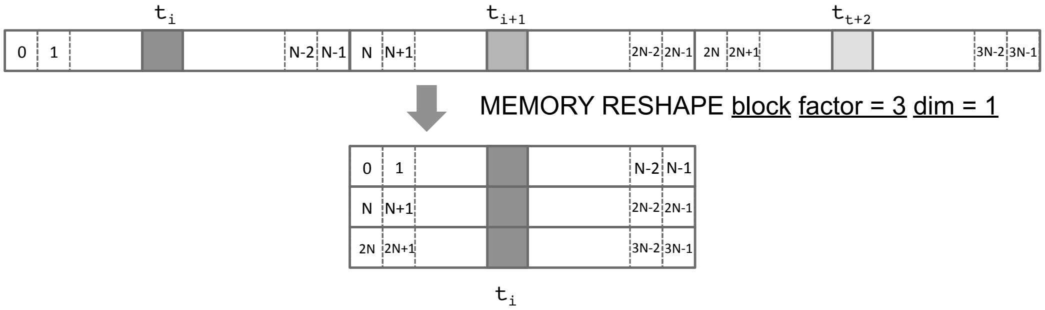


Figure 3.8: An example of memory reshaping. It folds memory block and re-aligns a data layout as user defined. It improves the memory port width and minimizes the memory bottleneck to load data.

Reshaping Memory reshaping is another method to maximize memory bandwidth. Figure 3.8 presents an example of memory reshaping. An on-chip BRAM can be reconfigured in the different data width. We can use a memory with a broader data width port for a same data using `#pragma HLS array_reshape` pragma. It partitions an array into multiple arrays like a memory partition does, but put into a single memory.

3.4 Image Processing on an FPGA

In this section, we will give an example of hardware design using HLS. One of the most common operations in image processing is sliding window, or image convolution operation. It is a two dimensional filtering for image matrix with a two dimensional kernel. Figure 3.9 describes how this operation processes. The input image height is H, and the width is W. The size of a kernel is defined as K, and in this example, K is 3. As the kernel window slides over image, it calculates a correlation between image pixels within the window and kernel coefficients.

A typical software version implementation for a sliding window is presented in Listing 3.1. It is based on a nested loop and accesses memory space as required. A HLS tool can synthesize this code as input and generates an RTL design as output. A

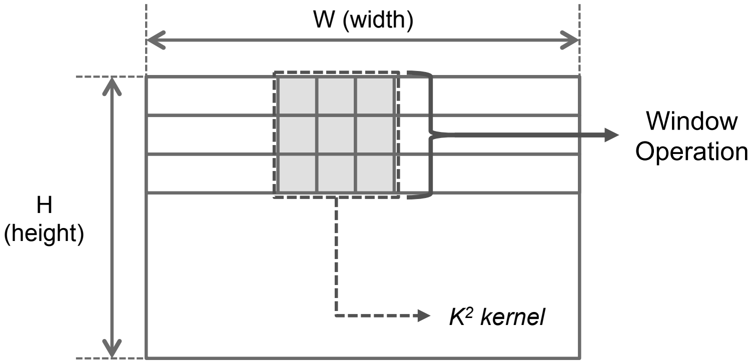


Figure 3.9: A convolutional window operation in image processing. The image width is W and height H. A $K \times K$ convolutional kernel slides over the image. This sliding window takes pixel values of image and calculates a convolution with the kernel coefficient. In this example, $K = 3$.

synthesized design with this code would hold input data and kernel values in a BRAM as long as the memory size is available. However, this type of nested loop implementation is not efficiently converted into a logic design. Pointing a target data space in a memory and taking data from it consume two clock cycles per data, and as the kernel slides over rows and columns, the for loop code should keep reloading data that is used in the past. In an FPGA design, we can arrange data and logic more efficiently and remove this redundant data access to memory.

```
void sliding_window_sw(dtype input[H][W], dtype output[H][W], dtype kernel[K][K])
{
    ...
    for(int j = 0; j < H ; j++ ){
        for(int i = 0 ; i < W ; i++ ){
            dtype sum = 0;
            for(int s = -K/2 ; s < K/2 ; s++ ){
                for(int t = -K/2 ; t < K/2 ; t++ ){
                    sum += kernel[s][t]*input[j+s][i+t];
                }
            }
            output[j][i] = sum;
        }
        }
        ...
}
```

Listing 3.1: A pseudocode for a convolutional sliding window operation in software implementation

We can easily observe the computational pattern of this operation. Window filtering operation takes neighbor pixel data points around a current point. So we have to utilize a temporal and spatial locality. Figure 3.10 presents an optimized architecture in hardware for the sliding window function. It uses line buffer and window buffer. A new input data firstly goes into line buffers temporally and shifts as data coming in. A window buffer is basically a set of registers that is fast accessible. The window buffer takes input image data pixels within a window and a user-defined kernel processes these data. The line buffer should be large enough to cache input pixels, and the window buffer is a kernel size number of registers.

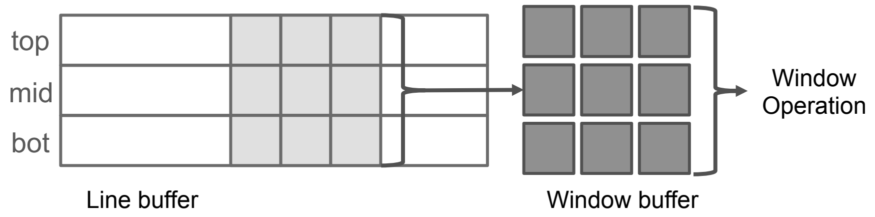


Figure 3.10: An optimized architecture for a sliding window operation using line buffer and window buffer. A line buffer is a BRAM block to hold incoming image pixel data in temporal. A window buffer is a set of registers partitioned to process window operation fast.

Listing 3.2 is a pseudocode of an optimized HLS implementation for the same window operation. As we described in Figure 3.10, it uses a line buffer and a window buffer. These line buffer and window buffer are defined as a single array for each in a code line 4 and 5. They are optimized using memory pragma, `reshaping` and `partitioning` in line 6 and 7 respectively. In this example, K is 3, so the line buffer is reshaped by factor of 3 to fold the data array. The window buffer is completely partitioned, and all elements in this buffer are synthesized as independent registers. Outer and inner `for` loops scan the image pixel by pixel. While a new input pixel comes (line 24), the window buffer shifts over the line buffer. A user defined kernel operation processes window data in line 26. At line 12, the pragma for pipeline operation pushes the design to run in pipelined way. When we synthesize this code block using Vivado HLS, a generated core can produce one pixel output pixel per one pixel input and achieve the maximum throughput (II=1).

```
1  #define K 3
2  void sliding_window_hw(dtype input[H][W], dtype output[H][W], dtype kernel[K][K])
3  {
4      dtype line[K][W];
5      dtype window[K][K];
6  #pragma HLS ARRAY_RESHAPE variable = line_buffer block factor = 3 dim = 1
7  #pragma HLS ARRAY_PARTITION variable = filter_window complete dim = 0
8      ...
9      for(int j = 0; j < H ; j++ ){
10         ...
11         for(int i = 0 ; i < W ; i++ ){
12 #pragma HLS PIPELINE II=1
13             ...
14             window[0][0] = window[0][1];
15             window[1][0] = window[1][1];
16             window[2][0] = window[2][1];

18             window[0][1] = window[0][2];
19             window[1][1] = window[1][2];
20             window[2][1] = window[2][2];

22             window[0][2]= line[top][x];
23             window[1][2]= line[mid][x];
24             window[2][2]= line[bot][x] = input[j][i];
```

```
            output[j][i] = kernel_filtering(window, kernel);
            ...
        }
        ...
    }
}
```

Listing 3.2: A pseudocode for a convolutional sliding window operation in hardware implementation

Again, the sliding window operation is a fundamental operation in image processing. For example, an interpolation for image resizing has a similar computation pattern except for the size of window and kernel operation. In our research in this thesis, we implemented different sliding window core modules and examined them. One of our module block, *blob search*, in Chapter 4 cascades several stages of sliding window with different kernels and input types.

As an example in Figure 3.10 and Listing 3.2, to build an efficiently optimized core design, an HLS code should describe an optimized hardware architecture and its behavior precisely using C syntax and pragma directives. We analyze patterns of other function modules and optimize them to achieve maximum throughput, or minimum interval in a system.

3.5 Conclusion

In this chapter, we briefly introduced an FPGA-based acceleration for our system design. Low-level hardware optimization on an FPGA is a non-trivial work. The design complexity is very high, and it requires in-depth background in logic architecture and design skills in system design. We presented optimization methods using High Level Synthesis tool. HLS enables designers focus on a behavior level optimization by rising the abstraction level in a system design. It hides all complicated process of low level optimization process and handles design constraints using a directive based API. However, it still needs some background for hardware design with application domain knowledge. So we introduced several key optimization methods on Xilinx Vivado HLS that we mainly utilized in our work in this thesis.

Chapter 4

Basic Hardware Accelerated Approaches

4.1 Introduction

Imaging flow cytometry is a high throughput cell analysis technique that can observe various 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 [28]. They are also useful for classification and sorting, e.g., mature stem cells can be separated from immature ones.

For this purpose, we introduced our target experimental setup in Chapter 2. It is designed to observe cellular deformability under a pressure generated by flowing fluid in a micro device, and a microscope-mounted high speed camera captures images at very high throughput. It processes more than 140,000 frames per second depending on a system setup.

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 such high frame rate. 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 2.4. 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. In our experiment, 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. In this chapter, we target to analyze these cell images in different hardware accelerator platforms, GPU and FPGA, and evaluate their performances comparing with the original CPU implementation. The target cell image in this experiment is very small, 32×208 pixel resolution.

Our primary contributions in this chapter 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 chapter is organized as follows. We explain our image analysis algorithm in Section 4.2. Section 4.3 presents hardware architecture optimization methods on an FPGA. Section 4.4 explains our GPU architecture. Section 4.5 presents our evaluation results, and we will conclude this chapter in Section 4.6

4.2 Cellular analysis

In this section, we will explain our basic cell image analysis algorithm and provide a high level understanding of it. There are four steps to perform the morphological analysis required for cell sorting. These are shown in Figure 4.1, and described at a high level in the following.

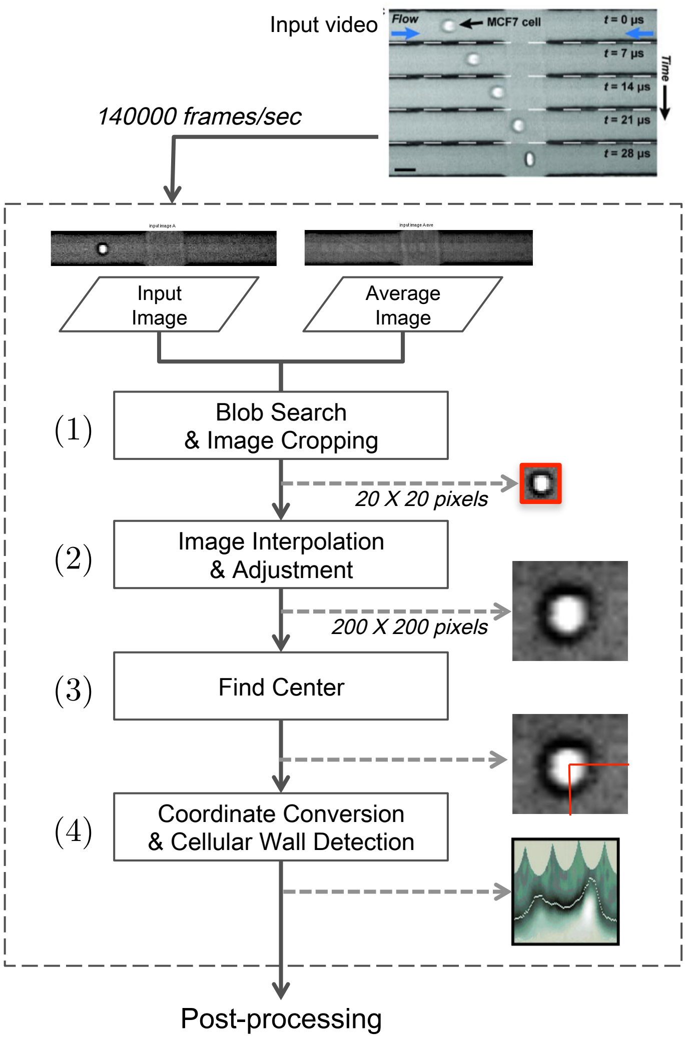


Figure 4.1: The stages of the image analysis algorithm include: (1) detecting the cell and cropping the area around it, (2) resizing the 20 × 20 cropped image into the 200 × 200 image and enhancing its contrast, (3) finding a center of the cell, (4) extracting morphological features by converting the image into polar coordinates based on the cell's center and cell walls.

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.

4.2.1 Blob search

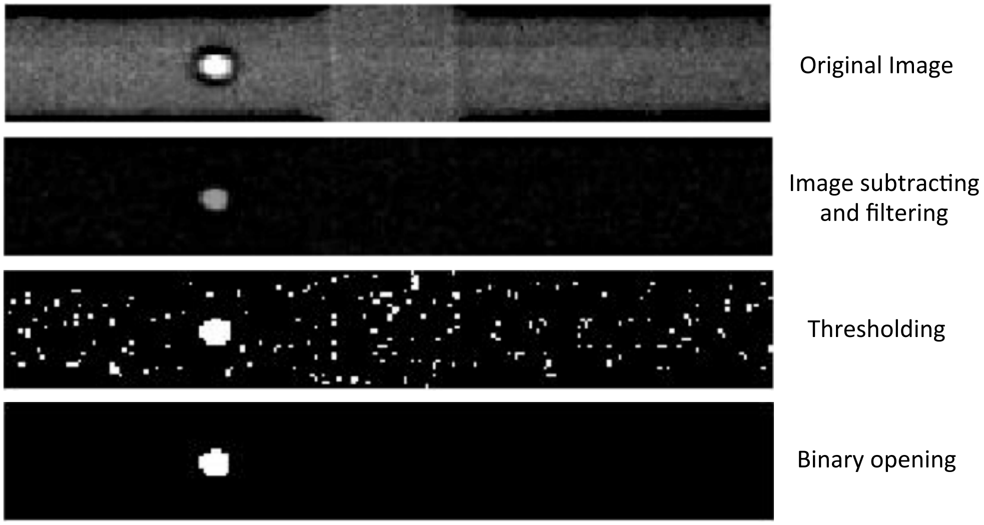


Figure 4.2: The *Blob Search* module performs background subtraction, thresholding which converts it into a binary image, and opening to remove noise. All these modules are based on an image convolution operation.

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 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 4.2).

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 20×20 sized region around the cell. The entire blob search process requires a single pass per image.

4.2.2 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 × 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.

4.2.3 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 4.3). 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.

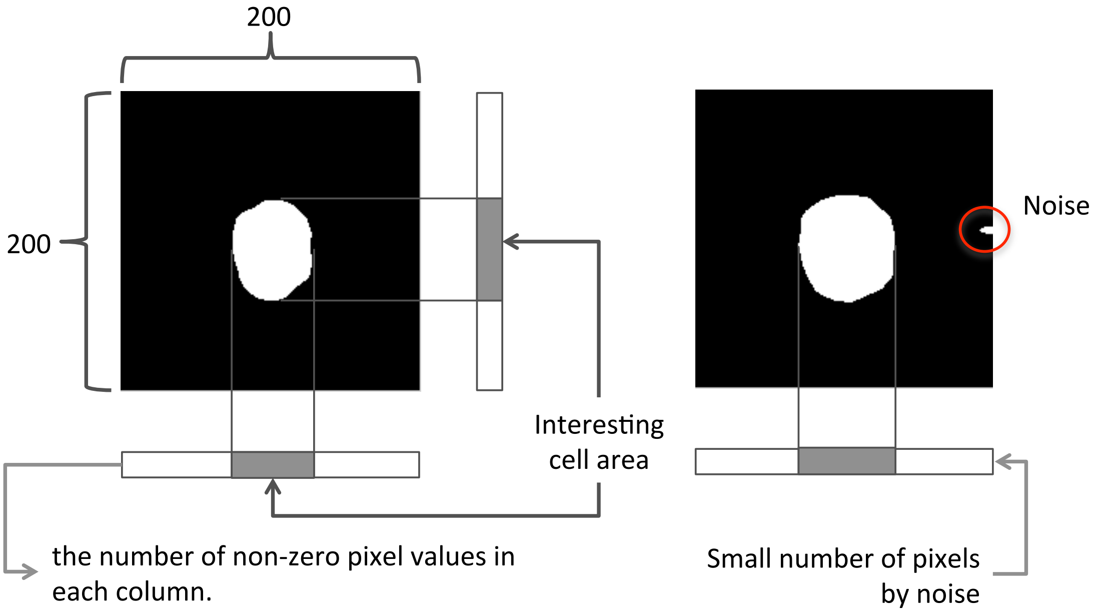

200
200
Noise
Interesting cell area
the number of non-zero pixel values in each column.
Small number of pixels by noise

Figure 4.3: The *Find Center* module performs binary thresholding on the interpolated (200 × 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.

4.2.4 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 4.6 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.

4.3 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

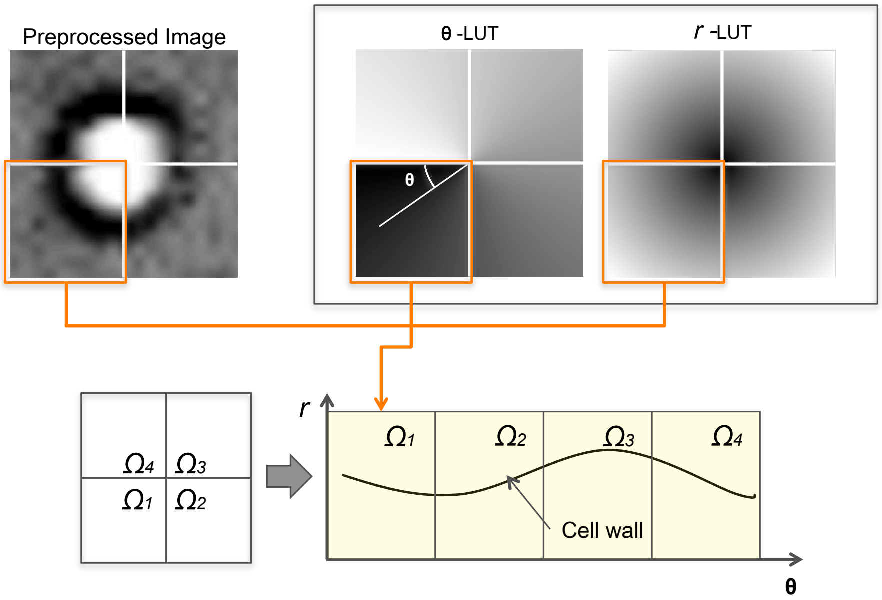


Figure 4.4: The coordinate conversion and radius extraction module. The coordinates of the cell wall are determined by scanning through the interpolated image to find the cell wall. This coordinate is then feed into two lookup tables that provide the corresponding angle and radius.

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.

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 4.5 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.

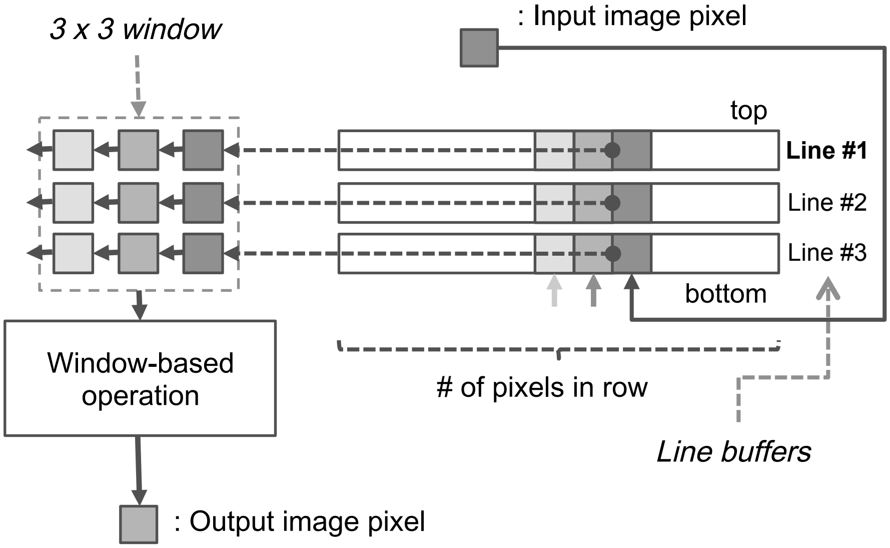


Figure 4.5: The architecture design for a shifting window operation on FPGA. New input pixels are stored in line buffer and a window buffer connected to it is used for window operation.

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.

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 4.6 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.

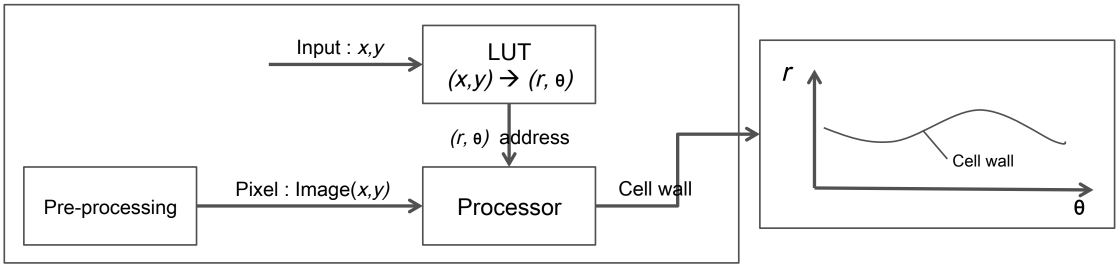


Figure 4.6: The architecture design for a coordinate conversion operation on FPGA. After pre-processing, an input image is converted into a polar coordinate based image.

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 4.9(c)).

4.4 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 4.7 (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 4.7 (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.

4.5 Experimental results

4.5.1 Experimental setup

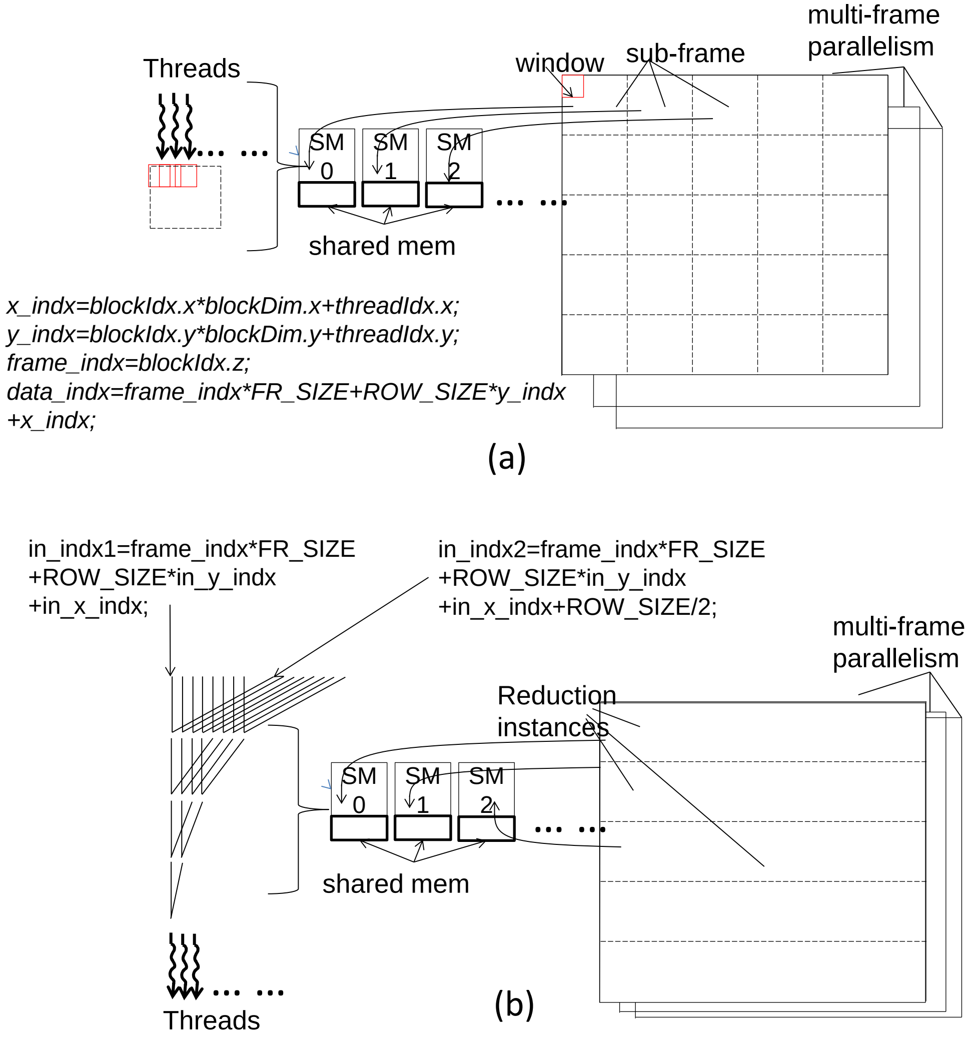
Our experiments use a 2.4 GHz Intel Quad Core Q6600 workstation for the MATLAB and C results. For the FPGA implementation, we targeted a Xilinx Virtex 6 (XC6VLX240T) using Vivado Synthesis Suite (Version 2012.2). The GPU implementation was tested using a NVIDIA GTX590 GPU with the CUDA 5.0 framework.

4.5.2 Results and comparison

Figure 4.8 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

Table 4.1: 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



```
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;
```

(a)

```
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;
```

(b)

Figure 4.7: 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.

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.

Table 4.1 shows the same optimized latency results including the number of clock

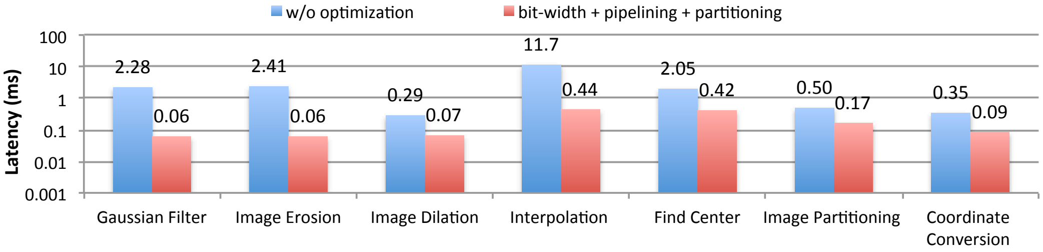
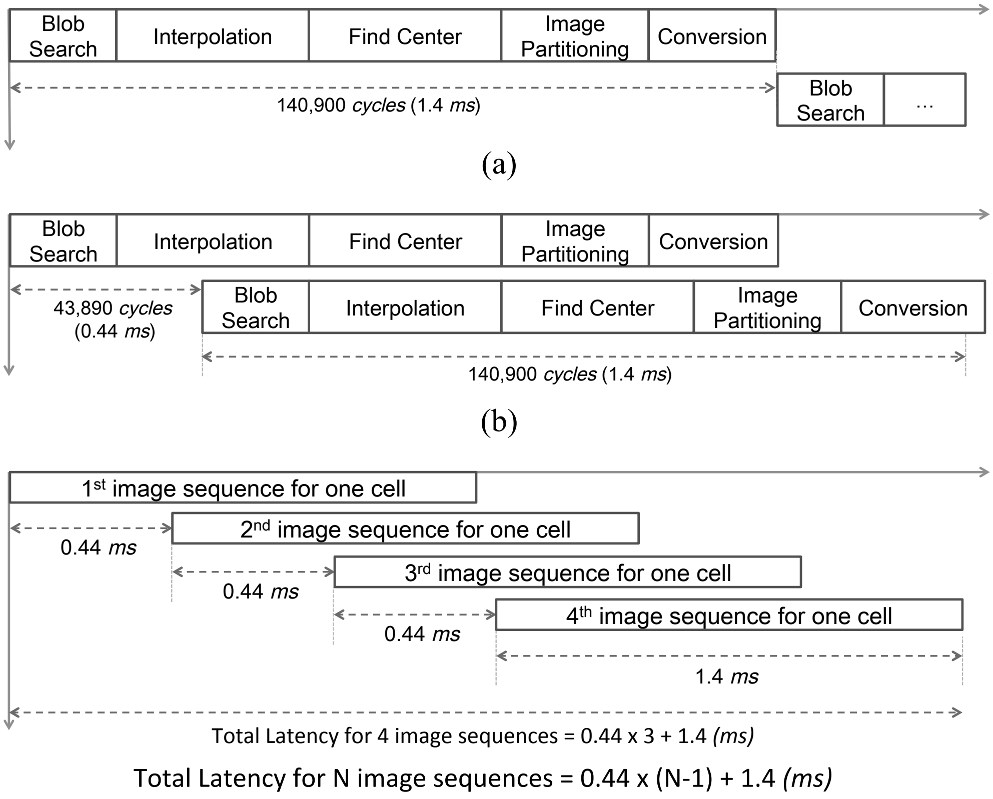


Figure 4.8: The performance of each module using different HLS optimizations in terms of latency(ms). After optimization in HLS, each functional module presents a huge performance improvement, more than ×3 up to ×40 faster in terms of latency.

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 4.9(a) 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 4.9(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 4.9(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 4.9(c).

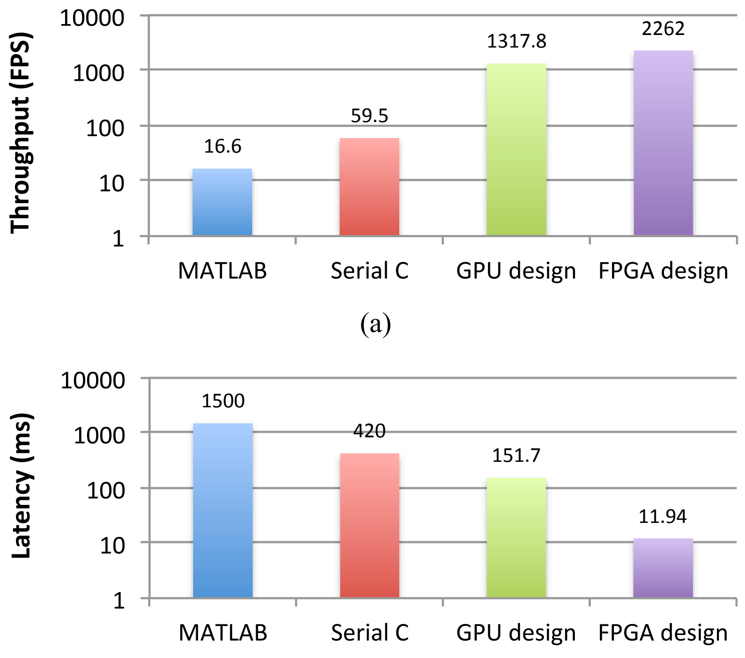


(c)

Figure 4.9: 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.

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).

Using Vivado HLS shortened our implementation time considerable as compared



(a)

(b)

Figure 4.10: 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.

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 4.10(a) compares the MATLAB, Serial C, GPU, and FPGA designs in terms of latency and throughput. Our FPGA design has approximately 38× more throughput (FPS) and 35× lower latency than the Serial C version. Our GPU has approximately 22× and 2.8× 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 4.10(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.

4.6 Conclusion

In this Chapter, we have examined our basic cellular image analysis method using different hardware accelerators, GPU and FPGA, and compared their performances. 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.

A GPU has been adopted in image analysis applications frequently, because it has high parallel computing power using many cores. In our experiment, it presets a good improvement in terms of throughput. However, an FPGA based acceleration is able to achieve better performance than the GPU design in terms of throughput as well as latency. A latency actually can be a critical issue for such a real-time system design. In the next chapter, we will present more advanced and optimized architecture on an FPGA for our target system.

This chapter, in full, is a reprint of the material as it appears in International Conference on Field Programmable Logic and Applications (FPL), Lee, Dajung; Meng, Pingfan; Jacobsen, Matthew; Tse, Henry; Carlo, Dino Di; Kastner, Ryan, 2013. There are small changes in format and phrasing as a chapter within this larger paper. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Advanced Morphological Analysis on an FPGA

5.1 Introduction

Quantitative analysis of cellular properties, such as size, shape, structure, life span, and molecular contents, can characterize cell function, give insight into how it behaves, and provide a technique for cell screening and/or sorting. However, there are strict performance constraints to achieve real-time cellular analysis; the system must have enough processing power to handle a very high cell throughput, and must performance cell feature analysis with a sub-millisecond latency to facilitate sorting. Our cytometry system is capable of analyzing thousand of cells per second based on image based technology, which corresponds to work at over 60,000 frames per second; this is a common goal in imaging flow cytometry [28, 2, 23].

A general purpose computer is not capable of achieving these performance, so an efficient computing support from hardware accelerators as coprocessor is necessary. GPUs and FPGAs are frequently suggested as a solution for such a high performance computing. They provide parallel or pipelined architecture in different granularity, and boost the performance utilizing processing elements in hardware level.

In the previous chapter, we evaluated different architectural approaches for our image analysis system on a GPU and an FPGA. Both approaches present considerable performance improvement over software implementation in terms of throughput and latency. However, in spite of a great improvement, a GPU-based approach does have a critical bottleneck in latency, which is one of main performance constraints, because of its memory bandwidth.

In this chapter, we explore more in-depth optimization on an FPGA and present an advanced cellular morphological analysis system. We target to achieve 60,000 frames per second throughput performance, corresponding to 2,000 cells analysis per second, and less than 1 ms for a single frame analysis performance. We adjust our target cellular image for 64 × 64 pixels per frame similarly tiny as our initial dataset. At same time, we expand our algorithm for more various datasets in different brightness level. They are still noisy and blurred, but different levels, so it is non-trivial to develop a system with high efficiency as well as accuracy.

We carefully develop a cellular analysis algorithm to extract their morphological features from microscopic images and build a real-time system using an FPGA device. There are various image analysis approaches for feature detection over low contrast images. However, these approaches are too computationally intensive and hard to achieve such high performance even with a hardware implementation. Most of them have iterative solution to refine analysis results or use spatial and temporal signatures to estimate target features accurately, which causes longer latency and lower throughput. Our method does not have iterative process to find a solution and minimizes data dependency for independent operations. It processes input and intermediate data in streaming way, which is intended for efficient hardware implementation in terms of performance and resources.

Our major contributions in this chapter are:

- Extend accurate image analysis algorithms for high speed cell morphological analysis.
- Hardware architectural optimizations using high-level synthesis (HLS) code.
- Developing a hardware accelerated system for microfluidic deformability cytometry.
- An in-depth evaluation and end-to-end demonstration of our system using a heterogeneous (CPU-FPGA) compute platform.

The remainder of this chapter is organized as follows. We explain our image analysis algorithms and hardware architecture optimization methods in Section 5.2. Section 5.3 presents the system description and experimental results in terms of accuracy and performance. We conclude in Section 5.4.

5.2 FPGA Implementation

In this section, we introduce our cell image analysis algorithm and its hardware accelerated architecture on an FPGA. To ease the design space exploration process, we design and optimize the hardware architecture using a high level synthesis tool, Xilinx Vivado HLS 2015.2. It allows us to focus on behavior level synthesis, data access patterns, pipelining, connections between modules, and so on, rather than low-level hardware debugging.

5.2.1 Overall flow

The imaging flow cytometry system is not only computationally intensive for image analysis algorithm, but it is highly data intensive. Our method processes these image data in streaming and fully pipelined manner in finer level as well as functional level for high performance and low resource usage. It has to handle massive amount of cellular images that are coming in the analysis pipeline in streaming way pixel by pixel initially. It can process incoming data when it is ready and forwarding it to next module right away with no storage required. It does not have iteration or feed data backward in its data flow, which prevents a faster pipelining operation. Our streaming data processing approach minimizes delay of analysis results and on-chip memory for caching data.

Also, modules do not have iterations. All functional modules can be implemented in one-pass processing, and input and intermediate data in the algorithm pipeline are delivered only forward. That enables us to create a hardware architecture that works on streaming images with a pipelined structure. The cell image analysis algorithm has two major parts: *cell detection* and *cell analysis*.

Figure 5.1 shows a cell analysis core architecture at a high level, corresponding to the algorithm flow. The *averaging* module generates a background image as a preprocessing step. It takes the first 256 frames and averages them. The averaged background is stored to a BRAM and persists in memory while the system is running. All incoming input images after the first 256 frames are directly connected to the *detection* module. The *detection* module is relatively smaller and simpler than the *analysis* module. It quickly checks if the current frame has a cell or not. The background image will be used in this process. Only valid cell frames are passed to the *analysis* module. The *analysis* module processes the rest of the major operations; deformation/morphological analysis. It consists of three stages; *find cell*, *find center*, and *trace cellular wall.* We synthesize these big modules separately and integrate them manually to generate a bitstream in Vivado 2015.2.

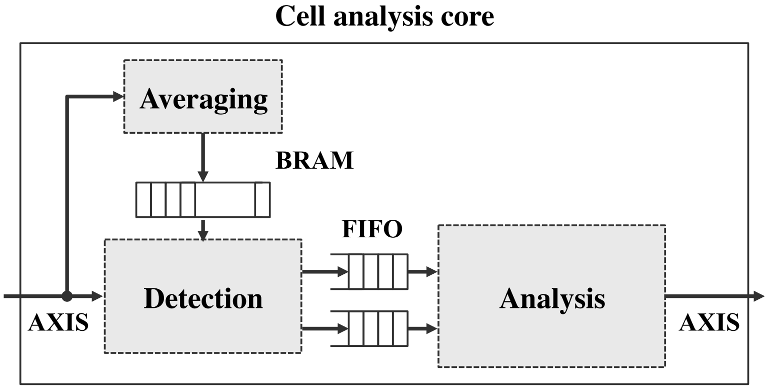


Figure 5.1: Cell anaysis core; *Averaging, detection, and analysis.* The averaging module takes the first 256 frames to generate a background image. The detection and analysis modules start running after that. The averaging module generates a background image and stores it using a BRAM, which is read by the detection module. The detection module passes intermediate images to the analysis module using FIFOs, and, if a current frame has no cell, it discards it.

5.2.2 Image analysis pipeline

Figure 5.5 shows a detailed block diagram of our hardware design including all modules and the connections between them. The box (a) in the figure is *cell detection*, and the rest of them, (a), (b), and(d), are substages of *cell analysis*, *find cell*, *find center*, and *trace cellular wall*, respectively.

Detection module

The *detection* module detects the presence of a cell quickly from incoming frames and passes only valid cell frames to the next stages, rejecting empty ones. We minimize the complexity of the *detection* module for a fast detection process. This rejecting process is based on a binary image, where it represents the cell area as white (or 1 in binary) pixel values (see in Figure 5.2 (a)).

In Figure 5.5 (a), when an input frame (C) comes in, it subtracts (B) the background image acquired by averaging the first 256 input frames in the *averaging* module. Then, it considers the bins in its histogram with the lowest intensity as background and selects a gray level value to separate the cell area from background. The binary image generated from this process may have noise in the background, so it applies a binary morphology operation, *erosion-only*, to leave only big particles, which is likely to be a cell. The number of valid true pixels in this frame is used to determine frames with a cell. An one-bit *iscell* flag indicates this frame has a cell (A).

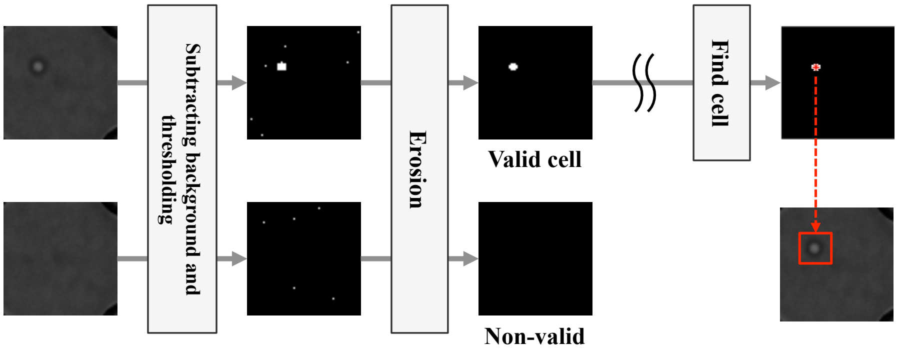


Figure 5.2: The cell detection process and the find cell stage in the cell analysis module (a)*cell detection*; it subtracts background from a given input image using thresholding. Then based on a converted binary image, it determines valid/non-valid cell frames.(b)*finding cell*; similar to *cell detection*, it finds a location of the cell from a denoised binary image, which is more accurate.

Find cell stage

The find cell stage needs a more accurate cell location than the cell detection stage. The input is the background subtracted image (B) from the *detection* module. Then, a Gaussian filter is used to denoise the input (E), and the thresholding module converts it (E) into a binary image. To remove extra particles from the background, it does a binary morphology operation, *opening*, i.e. *dilation* after *erosion*. The resulting binary image has a white blob on a plain black background representing the cell area as shown in Figure 5.2 (b). Averaging the number of these white pixels in each row and column gives an exact location of cell (D).

Find center stage

Based on the location of the cell (D), it crops a 24×24 cell area from three images (B,C, and E). The *resizing* module interpolates them 5 times and the *adjusting* module enhances its contrast. It converts the contrast-enhanced images to binary images to find the center point of a cell. In the binary images, white pixels represent the inner cell area or cellular walls as shown in Figure 5.3. It finds a center point (F) by averaging the number of these pixels in each row and column similarly to the *find cell* module. Averaging the 3 images also confers the benefit of reducing the noise since the errors in one are compensated for in the other images. The center point will be the input for the next stage.

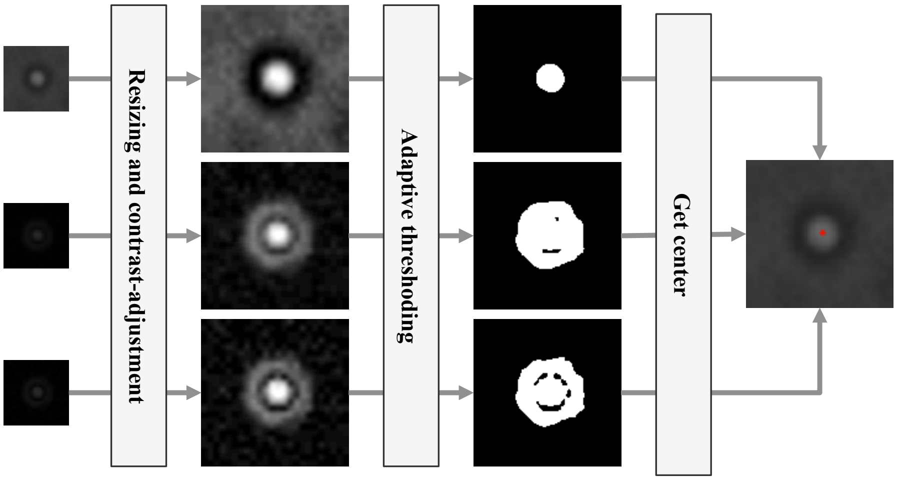


Figure 5.3: Find center stage; this stage resizes the cropped cell area from the three images based on the cell location found in the find cell module. Then, it resizes the cropped images 5 times and enhances their contrast. Adaptive thresholding converts the adjusted images to binary images. A center point is found by averaging the number of white pixels in each row and column.

Trace cellular wall stage

In this stage, the *conversion* module converts cartesian coordinate cell images into a polar coordinate images based on the center point (F). The horizontal axis represents the angle from the x axis in the original image, 0 to 360 degrees, and the vertical axis represents the distance from the center of the cell, or the radius. It uses the contrast-enhanced input image (G) as the module input and the darkest pixel in every single angle is considered the cellular wall. Finding the minimum intensity value at a particular angle is the simplest way to determine the distance to the cellular wall, but this method is likely to produce noisy results. So the conversion module extracts the distance to the cellular wall using several different methods and takes the median value of of the results for each angle.

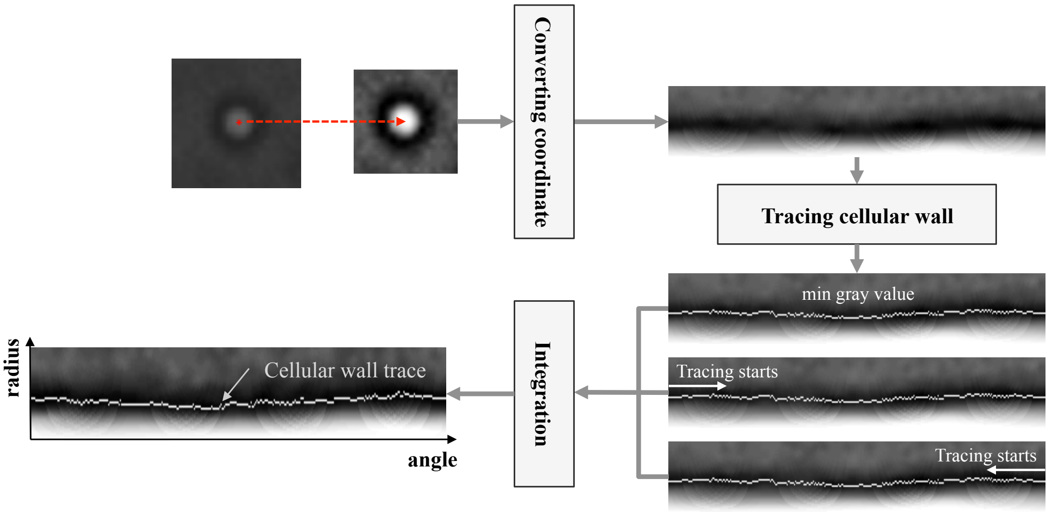


Figure 5.4: Trace cellular wall; The input to this module is the contrast-enhanced input image and the center point from the previous module. Based on the center point, it converts the cell image into a polar coordinate image and traces the cellular wall. The horizontal axis represents 0 to 360 degree angles. The vertical axis represents the radius, the distance from the center point to cellular wall in terms of the number of pixels. The lowest intensity values are considered to be cellular wall.

5.2.3 Hardare modules

Some modules in Figure 5.5 share some common patterns of computation or data access, like *templates*[49, 12]. While [49, 12] identify general computation or data access patterns, our work is more specific to image processing. The *sliding window* pattern is the

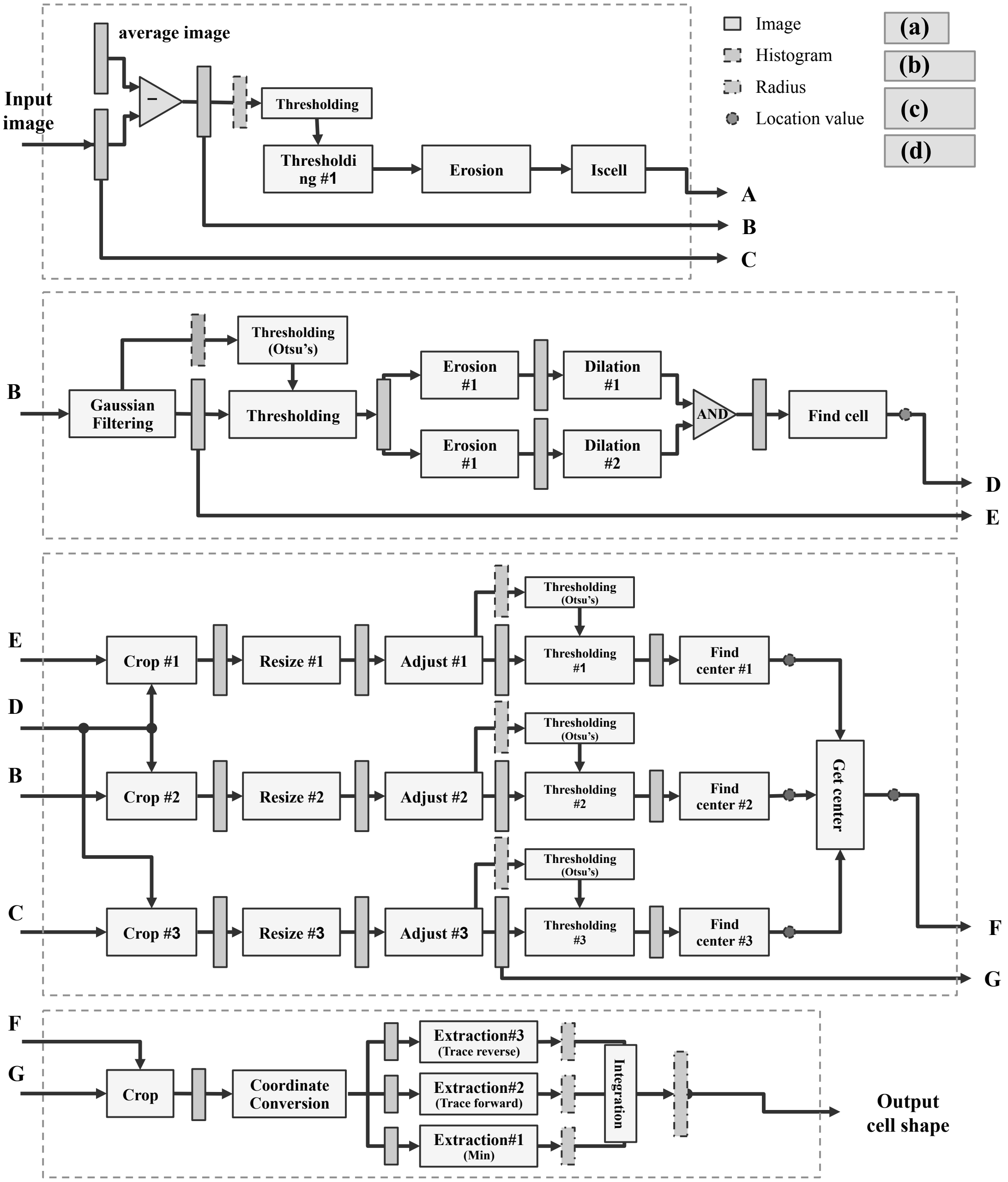


Figure 5.5: Cell analysis core pipeline block diagram (a) cell detection module (b)(c)(d) cell analysis module; (b) find cell, (c) find center, (d) trace cellular wall. The connections between these stages are noted alphabetically.

most frequently used pattern in this domain. Gaussian filtering, erosion, and dilation are sliding window kernel operations. Scaling is similar except it changes the size of output image. Thresholding and adjusting are matching a pixel value to another pixel value, and these are not dependent on neighboring pixels. The find cell and get center modules are group detection operations, which find a point by averaging the number of white pixels in a binary image in each row and column. Coordinate conversion transforms a geometrical shape of image, like warping, but also changes the size. Modules in each group are optimized in a similar way, and optimized modules are used to compose larger modules, such as *cell detection, find cell, find center,* or *trace cellular wall* in Figure 5.5.

Table 5.1: Modules grouped based on their computation patterns.

Groups		Modules
Sliding window	kernel operation	Gaussian filtering, erosion, dilation.
	scaling	Bicubic interpolation, cropping
	pixel-matching	Adjusting, thresholding
Detection		Find cell, get center
Geometry transformation		Coordinate conversion
Others		Generating histogram, Otsu’s method

This hardware design method combined with image analysis algorithm flow lets us estimate a deterministic performance result overall and gives a stable throughput in spite of probable algorithm flow update. Because they are divided into small modules and fully pipelined, only a bottleneck module decides the entire throughput. That means, even if the algorithm flow adds more modules for further extensive analysis, it does not affect to latency or throughput performance that much as long as additional modules are optimized to meet the similar condition within similar patterns, II in one clock cycle with a *pipeline* directive.

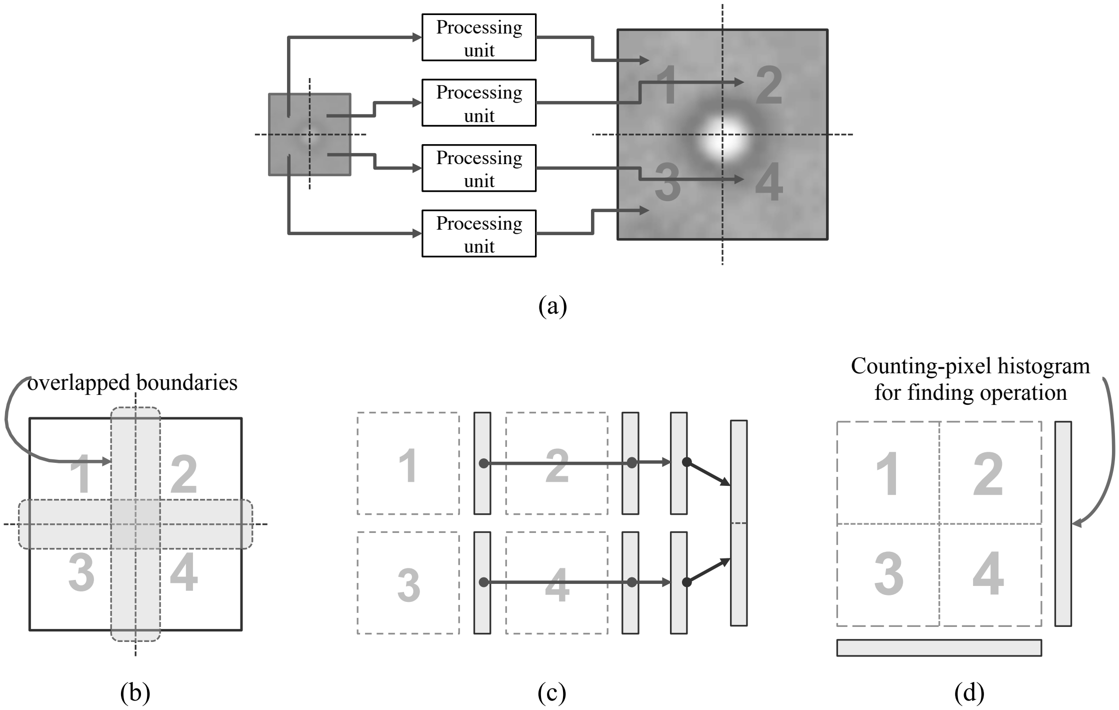


Figure 5.6: Hardware optimization for bottleneck modules; *Resizing, adjusting*, and *get center* are the main bottlenecks because they handle the largest size images. (a) To balance the overall performance, they are partitioned into four quadrants and parallelized. (b) The *bicubic interpolation* operation has weak data dependency on the overlapped boundaries region. If we ignore this, it can cause artifacts at the partitioned edges. (c) Generating a histogram has a data dependency on the entire scanned image. It generates a histogram for each partitioned image and reduces them. (d) The *get center* module also needs to scan the image by rows and columns. This process is also partitioned and reduced similar to (c).

5.2.4 Bottleneck modules

Our hardware implementation is fully pipelined at a fine-grained level (intra-modules) as well as a coarse-grained level (inter-modules). Most of the optimized modules achieve Initiation Interval *(II)* in one clock cycle using the *pipeline* directive. All submodules inside of two big modules run in a functionally pipelined way by applying the *dataflow* directive in HLS.

In a pipelined design, it is important to balance throughput performance by improving main bottlenecks since they are the critical path of the entire system. In our hardware design, the latency of each module depends on the size of image. The main bottleneck modules in the system pipeline are *find center* since it handles the largest frame after resizing; *resizing (bicubic interpolation), adjusting, thresholding,* and *get center*.

To achieve the higher performance, these modules should balance their performance with different modules. There are two ways to achieve that: scaling and partitioning. Scaling is simply replicating bottleneck modules multiple times and maximizing bandwidth, and partitioning breaks a bottleneck module down into submodules. Scaling is relatively simple to implement but it uses more resources to hold data for multiple frames. Partitioning should be done carefully considering data dependencies between submodules, which can introduce more complexity. But processing units in partitioning are smaller and use less resources. So we partition the bottleneck modules, *resizing (bicubic interpolation), adjusting, thresholding,* and *get center*, to multiple submodules that can run in parallel (see Figure 5.6 (a)). Each module operation and its data are divided and sent to four quadrants.

Bicubic interpolation

The *interpolation* module is basically a sliding window operation. Line buffers hold pixel data while a sliding window moves over them. It uses a 4 × 4 sized window for the bicubic operation. No temporal dependency between frames is needed, but there is a spatial dependency in one frame for neighboring window pixels. Ignoring this dependency can cause aliasing at cut boundaries (see in Figure 5.6 (b).) To minimize this error, line buffers should contain pixel values from the overlapping region. Cropped images are delivered to BRAMs and the line buffers can be filled from the cropped images.

Image adjustment

The *image adjustment* operation stretches an image histogram and matches each pixel to another pixel value. Generating the histogram requires checking every pixel value in a given image and therefore there exists a strong data dependency. To generate the histogram of the partitioned image without sacrificing throughput performance, it calculates small histograms for each quadrant first and then reduces them into one later (Figure 5.6 (c)). The histogram reduction is a one pass operation with a small input size, so it does not decrease the throughput performance.

Get center

The *get center* module is similar to the image adjustment operation. It also needs to scan the entire image *counting-pixel histograms* by rows and columns and has a strong data dependency (see in Figure 5.6 (d)). For partitioned images, each processing unit generates multiple histograms independently, then reduces them into one.

5.3 Experimental Results

In this section, we describe our experimental system and present the accuracy and performance results that we achieved.

5.3.1 System description

We tested our method on 3 different platforms: Matlab, C, and FPGA. The software implementations used a 2.3GHz Intel Core i5 with 8GB DDR3. The target FPGA board for hardware implementation is the Xilinx VC707, which has a Xilinx Virtex 7 FPGA device, xc7cx485tffg1761-2. We demonstrate an end-to-end system by connecting the FPGA board with a host computer communicating through PCIe. The control PC has Windows 7 or Ubuntu Linux and runs with the latest RIFFA 2.1 driver for FPGA communication [36]. On the software side, the host is responsible for reading data, streaming the data to the FPGA, and receiving the results (see Figure 5.7 (a).) The analysis core of the FPGA design uses the AXI-stream interface. This system is for offline analysis, and the connection can be replaced with any streaming interface for an online system.

5.3.2 Test dataset

The test data are organized in four different sets, which each have 5,000 frames. The frames are 64 $\times$ 64 in resolution, 4096-pixels, and stored in an unsigned 8-bit data type. The raw data are taken by a phantom camera [5]. Since this setup is very sensitive to light-level, the contrast level of the test video varies slightly. For accuracy testing, we take the first 1,000 frames and generate ground truth data. Then we compare our results to them (see Table 5.2.) The hit/miss ground truth data are manually produced, but others are from an initial work, which are also estimated results.

Table 5.2: Test video set for accuracy; the number of valid cell frames for *cell detection* results. The first 1,000 frames are taken from each video data. Note that the concentration of cells can be controlled by diluting the fluid.

	Set 1	Set 2	Set 3	Set 4
Cell	135	170	148	150
No-cell	865	830	852	850

The format of the input test data sent to the FPGA is in Figure 5.7 (b). The output data consists of the frame number, *iscell* signal, the location of the cell, the location of the cell center in the resized image, and 360-radius values (see in Figure 5.7 (c)). If the *detection* module decides a current frame has no cell, the *analysis* module doesn't start and generates no output. The first four bytes of the output, i.e. the frame number, help to synchronize an input frame and the output data by counting the number of frames sent to the FPGA.

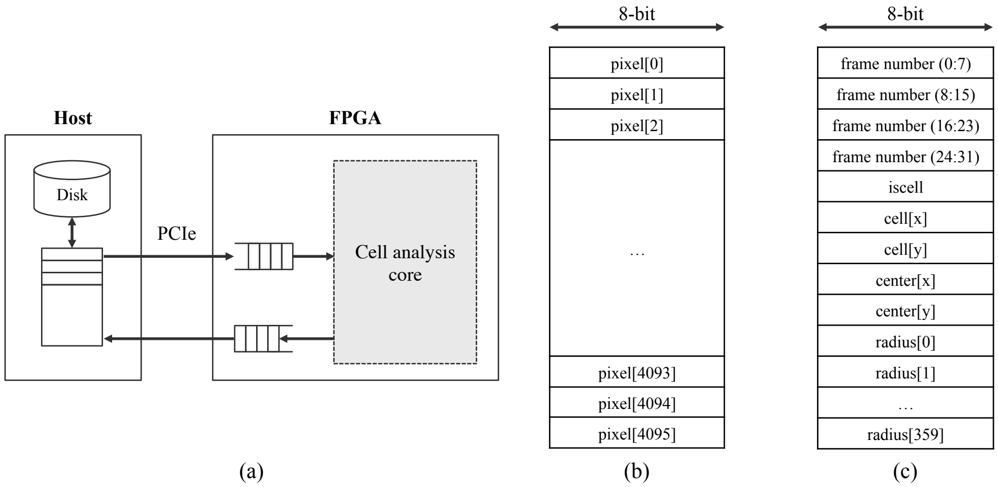


Figure 5.7: System description (a) offline cell analysis system connecting a host computer and an FPGA. The host reads raw data from a disk and sends them to the hardware using RIFFA for PCIe. All image analysis is processed on the FPGA side. (b)(c) input and output data format. Input data are streamed as 4096 pixel values and output data for a single frame consists of the frame number, valid cell flag (iscell), cell location from raw data, center point found, and 360 radius values in every angle.

5.3.3 Target throughput performance

Our initial target performance is to analyze 2,000 cells per second. If one cell event appears in 7∼15 frames and the event happens in 50% of frames, the system has to be capable of processing $28 \sim 60K\ frames/sec$ at the front end.

$$\frac{2000\ (cells/sec) \times 7 \sim 15\ (frames/cell)}{0.5\ (valid\ frames/total\ frames)} = 28 \sim 60K \quad fps$$

5.3.4 Accuracy results

For accuracy results, we use several metrics: (1) detection result (hit/miss) (2) find cell result (hit/miss) (3) size of cell (average radius) (4) respective ratio (ratio of long/short axis of cell.)

The *detection* output represents the result of determining cell/no-cell frames. The results in Table 5.3 show the true positive rate (*sensitivity*), true negative rate (*specificity*), ratio of true positives to number of positive predictions (*precision*), and true value rate (*accuracy*). They can be calculated as below.

- *Sensitivity*

  = (true positive)/(condition positive, or true positive+false negative)

- *Specificity*

  = (true negative)/(condition negative, or false positive+true negative)

- *Precision*

  = (true positive)/(test outcome positive)

- *Accuracy*

  = (true positive+true negative)/(total)

Table 5.3: Detection results with *sensitivity* (true positive rate), *specificity* (true negative rate), *precision* (ratio of true positives to number of positive predictions), and *accuracy*.

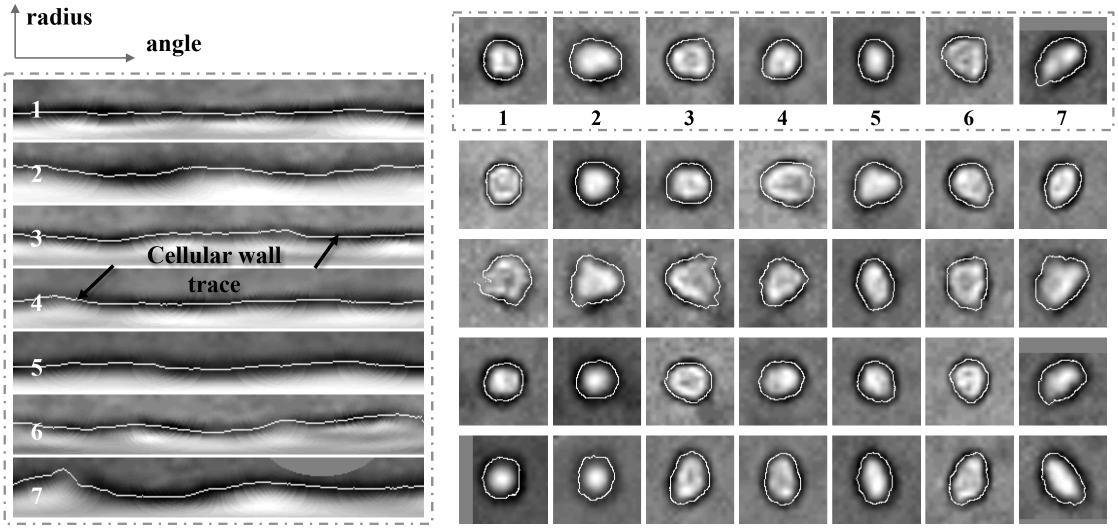
	Set 1	Set 2	Set 3	Set 4
Sensitivity (%)	55.31	66.47	64.86	75.33
Specificity(%)	99.88	99.64	100	99.29
Precision(%)	98.73	97.41	100	94.96
Accuracy (%)	93.60	94.00	94.8	95.7

Sensitivity means how many valid frames the system can detect out of true valid cell frames, and *specificity* means the number of empty frames the system can find out of true no-cell frames. And high *precision* represents that there is a high probability that most of the correctly predicted frames have real cell features. Our system ignores some frames at the edges or cell blobs that are too blurry, and effectively screens unnecessary frames and assures the validity of cell frames. A high *precision* result here indicates more efficient performance with fewer wasted operations.

Table 5.4: Find cell results representing hit/miss rate within a fixed distance from a true cell position.

	Set 1	Set 2	Set 3	Set 4
Cell location (%)	97.36	100	97.92	98.23

The Find cell result presents the correctness of cell location found. The next stage, cropping, cuts off cell focusing area based on this result. Table 5.4 shows the findcell result. It decides hit if a cell location is found within a certain boundary from a ground truth point, which is 5-pixels in Euclidean distance in this test.



(a) (b)

Figure 5.8: Trace cellular wall results example (a) polar coordinate images with the trace of the cellular wall in white lines (b) cell images with corresponding trace lines.

We check the size of cells and respective ratios to evaluate tracing cellular wall results. Figure 5.8 visualizes the tracing result examples using test cell images, and Table 5.5 shows mean absolute error (MAE), mean(μ) and standard deviation(σ) of cell size and respective ratios in each data set.

Table 5.5: Accuracy results in mean absolute error(MAE) and statistical distributions of test and ground truth data in terms of mean(μ) and standard deviation(σ).

			Set 1	Set 2	Set 3	Set 4
Size	MAE		1.31	1.37	6.88	1.74
	True	μ	15.03	14.73	14.20	12.43
		σ	2.52	2.71	3.26	2.98
	Test	μ	16.11	15.72	20.47	13.25
		σ	2.24	1.99	4.13	4.03
Ratio	MAE		0.22	0.17	0.26	0.21
	True	μ	1.04	1.03	1.19	1.00
		σ	0.29	0.29	0.36	0.35
	Test	μ	1.06	0.98	1.21	0.84
		σ	0.30	0.25	0.18	0.36

5.3.5 Performance results

In this section, we show our performance results on an FPGA and compare them to other design platforms. We have three different test platforms: Matlab, C, and FPGA.

Table 5.6: Throughput performance in detection and analysis modules of the hardware design pipeline.

	Detection	Analysis
Bottleneck latency (cycles)	4102	8287
Max clock frequency (MHz)	268	255
Max frame rate (FPS)	65.3 K	30.8 K

In the FPGA design, we build two main processes which run independently: *detecion* and *analysis*. Table 5.6 shows the maximum achievable frame rate of each module. The *detection* module deals with smaller images, so it spends less clock cycles than a bottleneck module and runs at a higher clock frequency. The *analysis* module consists of more submodules and processes more complex operations. The image size is larger in this module and the critical path is longer. Even though the maximum frame rate is less than *detection*, it doesn't affect the system performance since it is handling less frame data.

The performance of the entire design in terms of throughput and latency is

Table 5.7: Performance comparison in terms of throughput and latency for different platforms: Matlab, C, and FPGA.

	Matlab	C	FPGA
Throughput (FPS)	43.73	230.76	60.9 K
Speed up	×1392	×263.9	N/A
Latency (ms)	22.87	4.33	0.068
Speed up	×335.6	×63.6	N/A

represented in Table 5.7 across platforms. Latency is an average time to process one frame. Throughput is an inverse of latency and means the amount of frames processed in a second. The performance result of hardware is deterministic and predicable, which depends on the size of image, but software is not. Performance can differ across varying input. Our result is evaluated using the dataset introduced in Section 5.3.2.

When the FPGA design runs at a 250 MHz unified clock frequency, it is able to process 60.9 K frames per second at the front end. This is a ×1392 speed up when compared to the Matlab design and is ×263.9 faster than the C-based software design. The latency result also shows a significant speed up. The hardware design takes 0.068 *ms* to process one frame on average, which is ×335.6 faster than Matlab and ×63.6 than the C-design.

We compare our performance achievement with our previous work [42] in Table 5.8.. Even though the new image analysis process is enhanced and more complicated covering broader cellular image sets, it presents much faster throughput and latency performance against both FPGA and GPU implementations. Our new architecture filters valid cell frames quickly at the front end, and all bottleneck modules are balanced to achieve the target throughput as in Section 5.2.4. The new design results in about 30 times faster than an FPGA design in [42]. GPU design has a critical bottleneck in terms of latency because of data load operation. We utilize only on-chip memories for minimal latency, which is tiny but provides great memory bandwidth.

Table 5.8: Performance comparison in terms of throughput and latency with our previous work in [42]

	Previous works [8]		This work
	GPU	FPGA	
Throughput (FPS)	1.32 K	2.26 K	60.9 K
Latency (ms)	151.7	1.4	0.068
Frequency (MHz)	-	100	250

5.3.6 FPGA resource utilization

Table 5.9 shows the resource utilization results of our hardware design. The *detection* module consumes less than 1% of FPGA resources; 0.8% of LUTs and 0.3% of FFs. The *analysis* module uses 16.6% of BRAMs, 18.8% of LUTs, 7.72% of FFs, and 10.14% of DSPs. The entire design, including the PCIe API, consumes less than 20% of the resources overall.

Since the image analysis process runs with no dependency between frames, it can be scaled as much as the resource are available. The current design uses only 20% of resource on a target FPGA device, it is possible to add more processing element pipeline. If it scales the procedure multiple times, the frame rate could be more than the current 60K FPS up to few hundreds thousand frames per second. Our current goal in the algorithm is estimating the cellular feature accurately, but if the rest of resources could be also used for post processing after measuring cellular feature if necessary.

Table 5.9: Resource utilization in hardware design pipeline analysis. Utilization of *detection* and *analysis* modules and total utilization including PCIe connection.

	Detection	Analysis	Total
BRAM	0	328	390 (19.0%)
LUT	2653	45242	58533 (19.3%)
FF	1683	45338	60587 (9.98%)
DSP	0	299	299 (10.7%)

5.4 Conclusion

In this chapter, we developed and demonstrated a hardware accelerated cellular image analysis system. We designed an algorithm to analyze low resolution microscopic videos, and we created a custom hardware architecture to implement the algorithm on an FPGA. Our target setup is designed to extract cellular morphological features from a high speed camera. Our system meets the challenging performance requirements in terms of throughput as well as latency. Our system can handle video streams up to 60,900 frames per second and process each image with a 0.068 ms average latency. This provides the capability to perform real-time analysis and sorting of 2,000 cells per second.

This chapter, in full, has been submitted for publication of the material as it may appear in Journal of Parallel and Distributed Computing (JPDC), Lee, Dajung; Mehta, Nirja; Shearer, Alexandria; Kastner, Ryan. There are small changes in format and phrasing as a chapter within this larger paper. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Extensional Cellular Analysis based on Image Segmentation

6.1 Introduction

Image based cytometry technique enables sophisticated high contents cell analysis, and is capable of understanding complicated cell features that appears in a cell image. In the previous chapter we have used an efficient and accurate image analysis algorithm to determine the cellular morphological features such as size, shape, circularity, and deformability. In addition, this system can achieve high accuracy and throughput.

In the recent past, research works were focused on measuring physical shape of cells, however in this chapter, we would like to expand the cell analysis approach to understand different types of cell and its complex features. It includes observing the interior structure of the cell, such as membrane and nucleus. The cells can be irregularly shaped or extremely deformable, potentially exploding status in a device junction. We have explored several image segmentation approaches to evaluate these advanced features and different types of cells in detail.

Image segmentation approaches are commonly used in many application domains to understand objects in a scene. It basically partitions the image into chunks of the pixels, each of which represents an object or background. There are many image segmentation approaches in the literature. among which the simplest and easiest method is a binary thresholding approach. In this image based method, an object is separated from background depending on its threshold value and hence finding a proper threshold value is very important For more sophisticated advanced partitioning, clustering method based on classical machine learning algorithms is also well known. For example, *k*-means algorithm clusters the *n* number of one-dimensional pixel values into *k* clusters, and each cluster presents one object.

The primary motivation of this work is to analyze cell membrane and nuclei part or detect an irregularly shaped cells using image segmentation. We have explored several image segmentation algorithms in different categories. The first category is a binary thresholding based approach. The input cell data had very low contrast and was blurred, therefore it was non-trivial to find a proper thresholding value. We evaluated several algorithms to find a thresholding value accurately. In the second category, we used a data clustering based algorithm and *k*-means is the most commonly used algorithm for image segmentation. In our system, it clusters pixel values in an image into *k*-groups, each of clusters present cell nuclei, cell membrane, or inner cell architecture. In the third approach, we suggested the convolutional operation based segmentation in which we estimated the basic statistical property, mean or variance, in a convolutional window and cluster the pixel based on them.

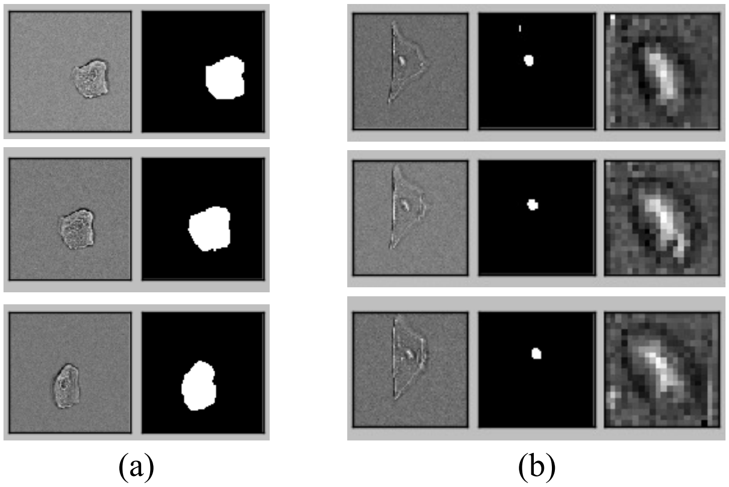


Figure 6.1: Examples of separating interior structure of cells. (a) Original input and segmentation result for cell membrane area. (b) Original input, segmentation result for cell nucleus, and nucleus only area *(left to right)*.

6.2 Methods

In this section, we have described several image segmentation algorithms that has been explored for cell image analysis. In general, we have examined various algorithms and picked five among them that presented desirable accuracy, performance, and hardware profiling results. We have categorized these five methods into three types based on their computation patterns. All the algorithms are carefully designed and modified to achieve our goal.

6.2.1 Thresholding based approaches

In general, the binary threshold based approach is considered as the simplest and easiest method for image segmentation that is used to separate a target object from its background depending on a threshold value. Technically, a current pixel presents an object, if it is greater than the thresholding value, otherwise, it is considered as a background pixel hence, it is very important to find a proper thresholding value. The value may be fixed and globally used for the entire image. The obtained value has to be adjusted based on spatial or temporal information for better segmentation results. The cell data that is obtained in our setup is very sensitive to light level and contains a lot of noise, so it requires an algorithm to adjust threshold operation. This research has explored various methods to determine the threshold value and to detect the target cell accurately.

$$out(x) = \begin{cases} object, & \text{if } in(x) \geq thresholding \\ background, & \text{otherwise} \end{cases} \tag{6.1}$$

Lumiance

According to the research works that have been explored, Luminance threshold method is the simplest method as it decides the thresholding value from a histogram of an image which can be described by probability density function. After the initial stages of preprocessing, such as enhancing contrast and removing noise, it calculates a histogram over intensity values of an entire image and finds a thresholding value to separate cell area. Figure 6.2 presents a process for luminance thresholding method. A thresholding value ($\mu + \sigma$) is a sum of a mean (μ) and standard deviation (σ) of pixel values in the current image and this value depends only on a current image, with no dependency between frames. Accordingly, each submodule can be processed by a single scan of the image.

Iterative selection

Iterative selection approach iteratively determines a thresholding value while scanning a current image. A thresholding value is initialized to a mean of image intensity values, and then it separates the cell area. Furthermore, it calculates a mean of background area T_b and cell area T_c separately and re-estimating a thresholding value as $T_b + T_c/2$.

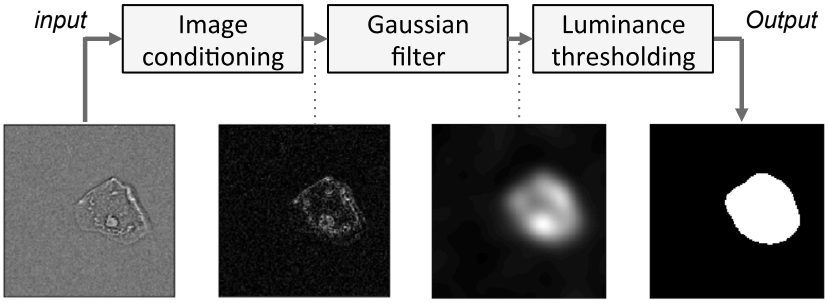


Figure 6.2: An algorithm flow of luminance thresholding method. It preprocesses input image and highlights the cell feature. Then, it filters the feature with Guassian filter and finds a threaholding based on a mean(μ) and standard deviation value(σ) of pixel values.

This process of re-estimating a threshold value is repeated until the preceding and the successive values converge. This gradual updating process refines a thresholding value and improves the final accuracy. Figure 6.3 presents an overall process for iterative selection method and its intermediate images between submodules. This method does not have a dependency between frames, that is temporal information. However, it does provide an iterative optimization solution to determine an absolute value. As shown in Figure 6.3, the final stage for iterative selection is a 2*N* passing process. *N* is the number of iteration to refine the value until it converges.

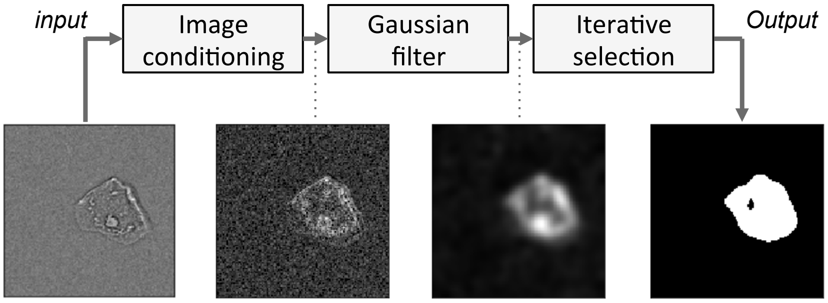


Figure 6.3: An algorithm flow of iterative selection method. The preprocessing and Gaussian filtering operations are similar as Luminance thresholding method, but it finds a thresholding value iteratively.

Hysteresis with a temporal signature

6.2.2 Data clustering based approaches

Data clustering is one of the fundamental problems in data analytics [60, 70], pattern recognition [65, 69], data compression [39], image analysis [59, 53], and machine learning [14]. It groups data objects into several subsets known as clusters, based on either their similarities or dissimilarities, where data items with distinct features are categorized in separate clusters. It is widely used to analyze given data set and understand its general features.

In computer vision or image analysis domains, it is generally used to understand visual information and extract object features from images such as segmentation, object detection, or object recognition. It considers a single pixel, a chunk of pixels, or an entire image as input and infers what a data object unit represents in the image. We have examined the cellular image data set using *k*-means, which is most frequently used algorithms for image segmentation.

k-means algorithm

In this section, we have tested k-means clustering algorithm. *k*-means partitions n input data, x_i's, into k groups, $\{X_1, X_2, ...X_k\}$. Each of them is presented using a *single* center point, c_i. The algorithm assumes that data points from d-dimensional spherical density shape are centered on this mean point. *k*-means algorithm finds an optimal set of center points that minimizes an objective function in (6.2). It is defined as the sum of distances over centroids and data points associated.

$$\underset{X}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x \in X_i} ||x - c_i||_L \tag{6.2}$$

To find an optimal solution, *k*-means uses an iterative refinement process that performs an assignment of the data objects to clusters and subsequently updates the centroids. The centroid update would result in a new assignment of the data objects. i.e., This process continues until the assignments converge, The clustering problem is *NP*-hard, and this algorithm does not guarantee an optimal solution. Nevertheless, its simplicity and intuitive nature have led to a widespread use of partitioning across a wide number of applications domains.

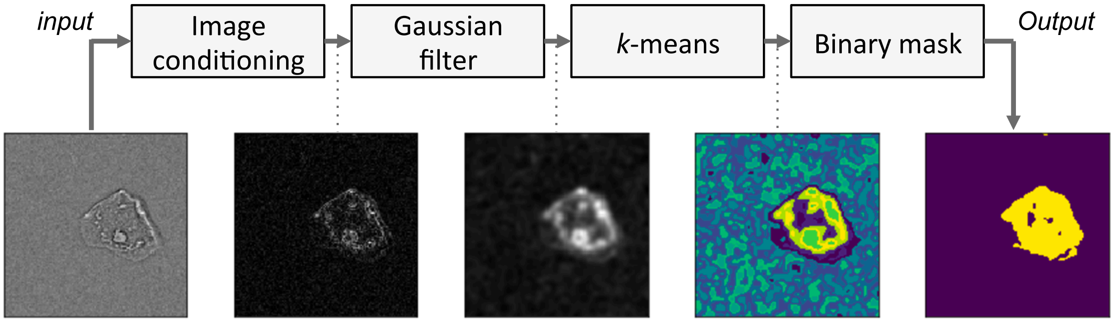


Figure 6.4: An algorithm flow of *k*-means based approach. This method partitions image pixels using *k*-means clustering algorithm. Then, it defines the largest pixel region as a cell membrane area.

Figure 6.4 presents the *k*-means based image segmentation approach. In our test, we have initially applied image conditioning and Gaussian filtering for preprocessing and *k*-means algorithm is used to cluster pixel points.. The number of clusters, *k*, is set for 10 in this test. If *k* value is larger, it can cluster finer data finer, that is higher clustering resolution but would take more time to find a solution. After the process of clustering, the average intensity of the pixels in each cluster is verified and a binary mask is created to select cell area based on the total number of pixels present within in each cluster. Figure 6.5 shows an intermediate image data for different frames in a single cell event.

6.2.3 Convolutional window based approaches

Convolutional window process, or sliding window process, is the most frequently used data processing pattern in image analysis application. It is employed in filtering,

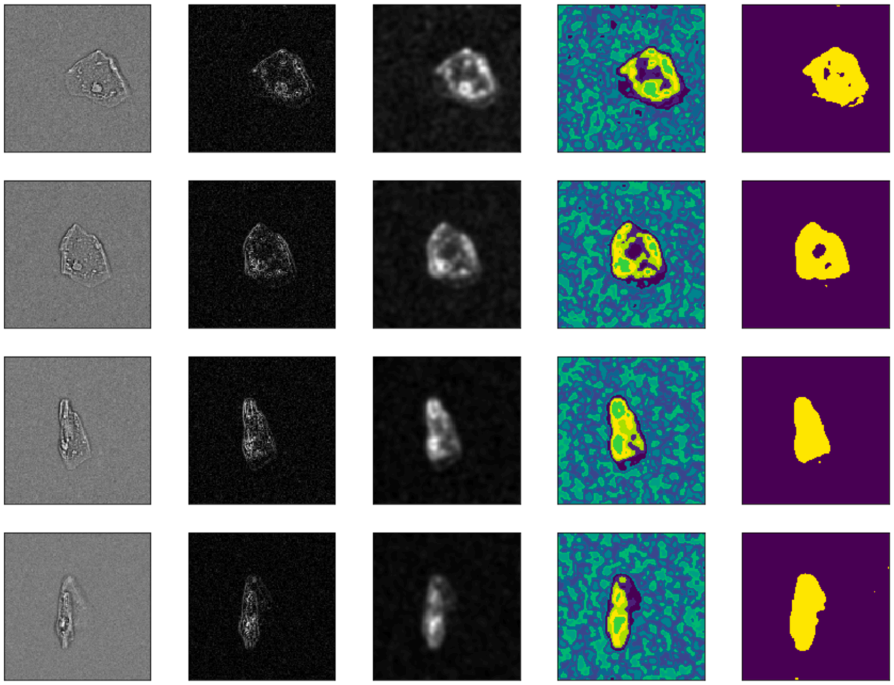


Figure 6.5: Examples of intermediate image data in *k*-means based segmentation approach

denoising, edge detection, correlation, compression, deconvolution, simulation, and in many other applications. A kernel operation within a convolution window is diverse. A kernel is a (usually) small matrix of numbers that are used in image convolutions. Differently sized kernels containing different patterns of numbers produce different results under convolution and size of a kernel is arbitrary but 3×3 is often used. Most of our module operations in Chapter 5 are similar. A convolutional window based operation is one-pass processing algorithm and very hardware friendly. It can be easily implemented in a hardware using line buffers ,window registers and is able to achieve high throughput with a low latency.

This operation takes a chunk of pixels area within a window and uses neighboring data properties, not only by pixel-by-pixel. We consider statistical values, mean and variance within a convolutional window. This convolutional approach started from an idea that a window near the cell area has brighter intensity (mean) or higher complexity (variance) than the background.

Convolutional mean

The Convolutional mean method is same as a 3×3 mean filter. and is commonly used for noise removal. In our cell data set, the pixel data is a low intensity , bright pixels along with cell edge or nuclei. So, this mean filter smoothes out the cell image except for nuclei and cell edges as shown in Figure 6.6 (b).

Convolutional variance

Convolutional variance method calculates the variance of a 3×3 sliding window. Background pixels are highly noisy and blurred, but very consistently. It does not have much difference with the neighbor pixels. However, cell area has a higher complexity that is high variance. In this method, it is possible to catch a transparent cell within the fluid, which would have a similar level of intensity range to that of the background having only subtle noises in a fine level (shown in Figure 6.6 (c)).

Choosing thresholding value

In both methods, we have chosen a threshold value over convolutional mean or variance image to decide whether a pixel is a cell or background. From the data set it can be noticed that the characteristic, size, and features of a cell varies over frames and making it difficult to pack a single parameter value for thresholding. Therefore, we have combined convolutional mean and variance method and utilized distribution property to choose the parameter for cell area.

We have used a standard normal table, called Z table, which represents the cumulative distribution function of the normal distribution. The thresholding value was determined based on the anticipated cell size. For example, if the cell size is large enough

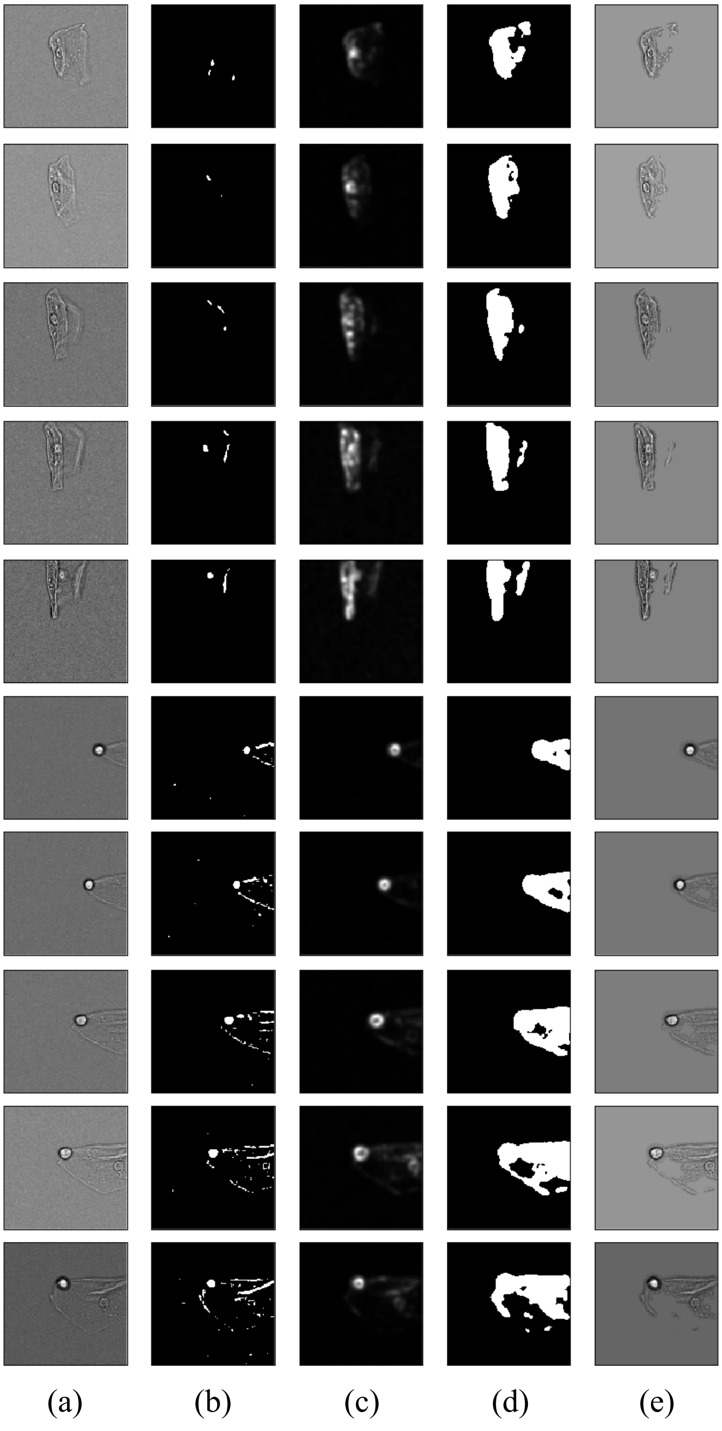
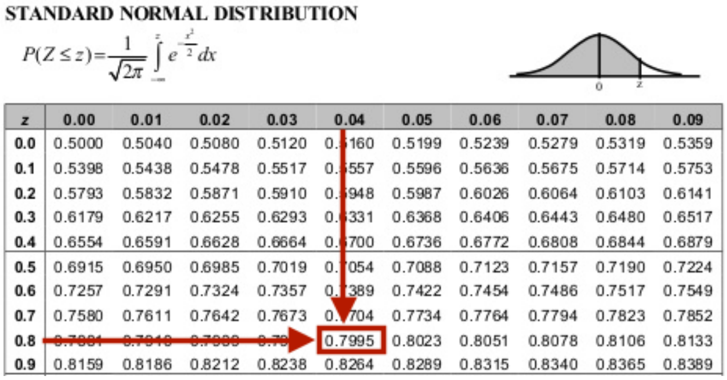


Figure 6.6: Examples of preliminary test images for convolutional window based approach (a) input images (c) thresholded images after applying a convolutional mean operation, (c) images after applying a convolutional variance operation (d) thresholding of (c), (e) cell-only images by applying a binary mask in (d).

to cover more than 20% of pixels, we can mathematically estimate the thresholding value that presents a top 20% point in pixels distribution, i.e. histogram, of a convolutional variance image.

Parameter estimation from a standard normal distribution

A parameter from standard normal distribution has to be estimated. Every normal distribution is a version of the standard normal distribution whose domain has been stretched by a factor σ (the standard deviation) and then translated by by μ (the mean value). In addition, any normal distribution $f(x|\mu,\sigma)$ or standard normal distribution $f(z|0,1)$ can be converted vice versa.



STANDARD NORMAL DISTRIBUTION

$$P(Z \le z)=\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z} e^{-\frac{x^2}{2}}dx$$

z	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
0.0	0.5000	0.5040	0.5080	0.5120	0.5160	0.5199	0.5239	0.5279	0.5319	0.5359
0.1	0.5398	0.5438	0.5478	0.5517	0.5557	0.5596	0.5636	0.5675	0.5714	0.5753
0.2	0.5793	0.5832	0.5871	0.5910	0.5948	0.5987	0.6026	0.6064	0.6103	0.6141
0.3	0.6179	0.6217	0.6255	0.6293	0.6331	0.6368	0.6406	0.6443	0.6480	0.6517
0.4	0.6554	0.6591	0.6628	0.6664	0.6700	0.6736	0.6772	0.6808	0.6844	0.6879
0.5	0.6915	0.6950	0.6985	0.7019	0.7054	0.7088	0.7123	0.7157	0.7190	0.7224
0.6	0.7257	0.7291	0.7324	0.7357	0.7389	0.7422	0.7454	0.7486	0.7517	0.7549
0.7	0.7580	0.7611	0.7642	0.7673	0.7704	0.7734	0.7764	0.7794	0.7823	0.7852
0.8	0.7881	0.7910	0.7939	0.7967	0.7995	0.8023	0.8051	0.8078	0.8106	0.8133
0.9	0.8159	0.8186	0.8212	0.8238	0.8264	0.8289	0.8315	0.8340	0.8365	0.8389

Figure 6.7: A Z table for normal distribution. A thresholding value can be estimated by the expected area rate (%) and this table.

For example, to find a threshold value cutting off upper 20% in a given data set, which is assumed to be in a normal distribution, $f(x|\mu,\sigma)$, can be estimated from the standard normal distribution, $f(z|0,1)$. The point of the threshold value for upper 20% in a standard normal distribution is already defined in a standard normal table, which is about 0.84(as shown below). From the fact, $P(z > 0.84) = P(x > \alpha)$ From a conversion equation above, we can find the thresholding value $(\alpha - \mu)/\sigma = 0.84$, $\alpha = 0.84\sigma + \mu$. The procedure to find the parameter 0.84 from the table given below. This kind of table consist all z values and a probability of $P(z < Z)$ for each z. From a left column, 0.8, and from a top row, 0.04, a cross section, which makes 0.84, has 0.7995. This indicates the probability of an area below 0.84 is 0.7995 in the standard normal distribution.

Advanced convolutional mean and variance method

Figure 6.8 presents an algorithm flow of our convolutional segmentation approach. It consists the following steps: (1) generating a convolutional mean and variance images by using a 3×3 convolutional kernel. (2) estimating a cell size in a simple method. We set upper and lower bounds in a noise removed image and count the number of active pixels roughly expected to be a cell area. (3) Based on the expected size, thresholding parameter in variance image can be estimated and a standard normal distribution is used at this stage. (4) It separates the cell area and background based on these parameters.

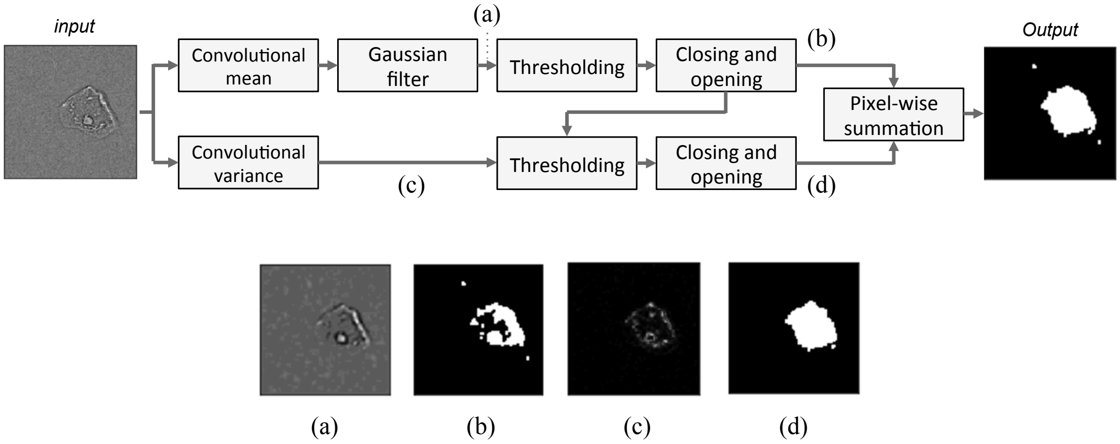


Figure 6.8: An algorithm flow of convolutional window based approach. (a),(b),(c), and (d) present intermediate image during the process.

6.3 Experimetal Results

In this section, we evaluate different image segmentation approaches in terms of accuracy and performance.

6.3.1 Test environments

Our test image data examples are in Figure 6.9. Size, shape, structure, deformability, and its transparency are various. We have 319 cell images with 27 different cells. The estimated cell area is measured by the number of pixels within a cell region and compare to our ground truth data. We test the accuracy and computing performance in python software.

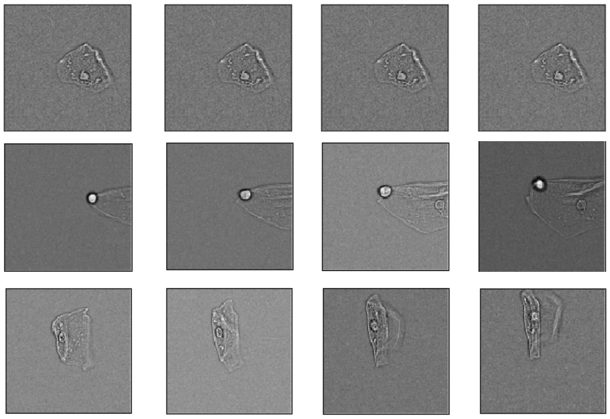


Figure 6.9: Examples of test image data. Each row presents for a single cell. Cells are in different shape, structure, and transparency.

6.3.2 Test results

Figure 6.10 shows accuracy of different algorithms. It presents the number of error pixels in each segmentation result and an error rate on the top. All of them show less than 4% of error rate. Luminance thresholding method presents the most error (3.44%), which is the simplest approach. And iterative selection method gives the best accuracy result (2.86%).

Figure 6.11 presents a comparison of computing time in software. As we discussed, the simplest luminance thresholding method gives the minimum computing time (0.8 sec). *k*-means algorithm contains iterative process, so it gives high latency

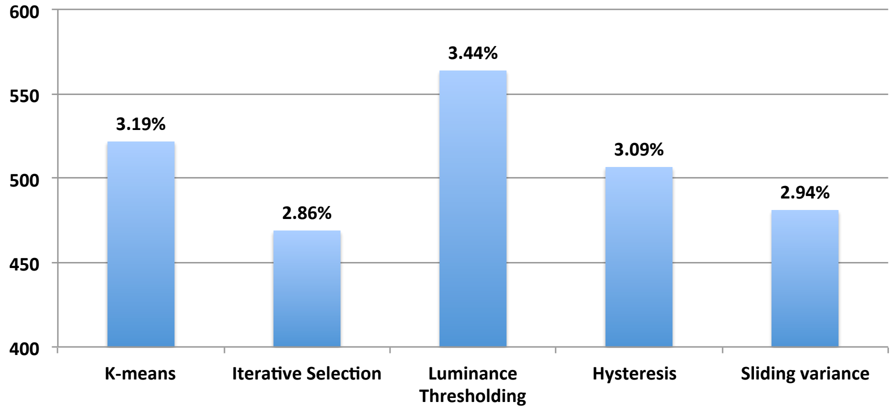


Figure 6.10: Accuracy of different segmentation algorithms. It is based on the number of pixels of cell area difference (vertical axis). Each result presents an error rate on the top.

(6.3 sec). Hysteresis with temporal signiture method gives also high computing time (5.1 sec) because it refers extra frames before and after the current one. Convolutional method gives the highest latency overall. It moves a convoultion window over image pixel by pixel and calculates variance within the window. It takes up to several minutes to calculate a mean and standard deviation for every convolutional window.

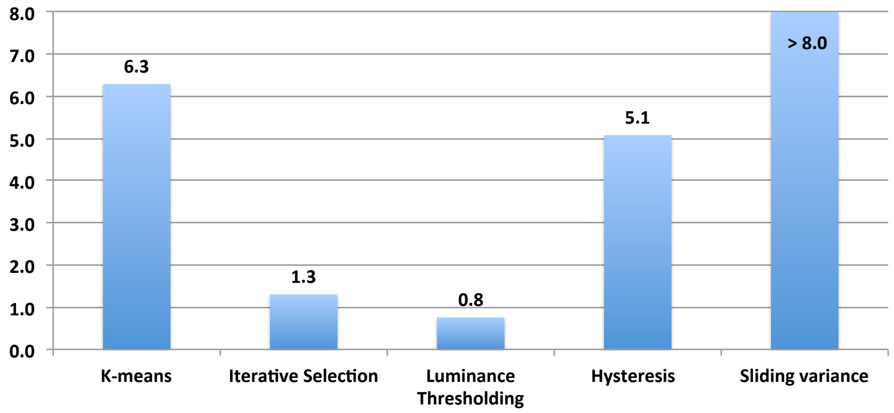
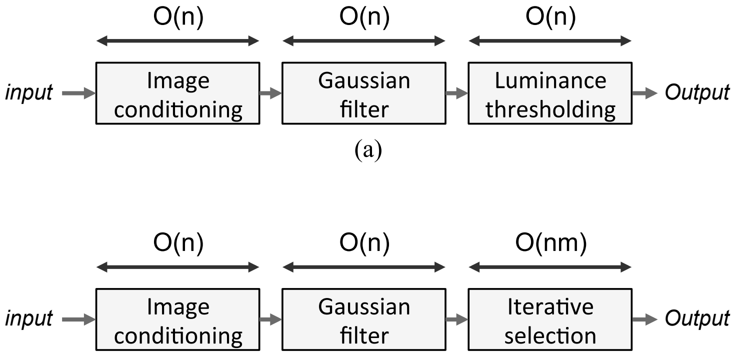


Figure 6.11: Latency of different segmentation algorithms in software. Luminance thresholding presents the minimum latency, and convolutional variance method takes a few minutes to calculate variance values in sliding convolutional window.

6.3.3 Design complexity and hardware implementation

Since we are targeting a highly pipelined hardware implementation mostly running in streaming, the data bandwidth is a main bottleneck and decides the overall performance. Therefore, a module that handles the biggest image is a bottleneck in general. We estimate the most achievable performance based on the algorithm complexity in their bottleneck module (see Figure 6.12).

When the size of image handling is n, luminance thresholding method has $O(n)$ complexity. Iterative selection has $O(nm)$ complexity when m is the number of iteration to converge results. k-means algorithm is already known as NP-hard problem. When the number of iteration is fixed to i for a simpler case and k is the number of clusters, it is $O(ikn)$. Convolutional variance is $O(n)$. Based on this fact, the algorithm that can be implemented the most efficiently on hardware is luminance thresholding method or convolutional variance.



(a)

(b)

Figure 6.12: Examples of complexity analysis in an algorithm flow. The most complex module is the bottleneck in a pipelined system. (a) luminance thresholding method is balanced (b) iterative selection method has a bottleneck in the last stage.

6.4 Conclusion

In this chapter, we have discussed different image segmentation approaches for cellular image analysis and have explored thresholding based approach, data clustering based approach, and convolutional window-based approach by using luminance, iterative selection, hysteresis with temporal signature, k-means clustering, and convolutional window based methods. Luminance thresholding is the simplest method that presents the minimum performance overhead in software and is hardware friendly. However, it tradeoffs the accuracy and does not provide an absolute thresholding value per frame. The Iterative selection consists of a refining process to find a thresholding value and provides more accurate segmentation result but hinders high throughput performance. Hysteresis method with a temporal signature does generate a good estimation of the cell area, while the frame dependency causes a longer latency, which is not suitable for hardware implementation. Convolutional window based approach provides a good estimation for cell area and has a promising performance for high throughput analysis in a pipelined way on a hardware. However, regarding our primary goal to observe more detailed inner cell information, we can conclude that a data clustering based approach is the best choice for that. In the next chapter, we will present our work about the streaming data clustering algorithm.

Chapter 7

Streaming Clustering Algorithm for Image Segmentation

7.1 Introduction

Microfluidic cellular images provides indepth information of cells. We can observe cellular features in detial and extract complicated parameters from images in high throughput. In our previous works, we measure cellular morphological features, i.e. external shape of cell membrane [42], and observe physical structure and deformability under stress in a device. We want to extend our system's capability for more functionality. Cell has more complicated inner structure, such as separable cell nuclei and membrane, or sometimes its shape is highly irregular to define. For this purpose, we need more sophisticated method than a binary thresholding or measuring its shape. Data clustering based image segmentation approach helps us to analyze and understand cell images with these details.

Image segmentation is one of the most common and classical problems used to detect objects and understand scenes in computer vision area. It partitions an image into multiple regions, each of which may presents an object or a background. There are many practical application using this method and research works as well to find a general solution for that. Data clustering based segmentation is one of well-known solution. It takes an image pixel value or a set of pixels as an input data point and generates sets of pixels as a clustering result. In our cell analysis system, it enables accurate and detailed analysis of cells.

However, finding a solution has a complicated optimization process, which is commonly very iterative to refine the solution. This iteration holds strong data dependency, and it hinders designing a high performance segmentation applications. In our system, we have two critical performance constraints in throughput and latency. Even using a state-of-art accelerator hardware, achieving these performance goals is a non-trivial problem.

In this chapter, we present our effort of designing an accelerated data clustering system on hardware for image segmentation. We develop a novel hardware friendly data clustering method and accelerate the algorithm on a CPU-FPGA heterogeneous system. Our final hardware design achieves high throughput performance with reasonable resource utilization, which enables it to scale towards large and high dimensional data sets. We evaluate our system for image segmentation with our cellular data set as well as other open sourced bigdata sets for broader applications. Our clustering method outperforms the state of the art clustering algorithms in software system [10] and FPGA implementations of heterogeneous systems [46, 64, 8].

Our primary contributions of this research are:

- Suggesting a high throughput image segmentation method for analyzing more detailed cell analysis
- A hardware friendly, multilevel, streaming clustering algorithm that can handle large, high dimensional data sets

- A hardware/software codesign method for streaming clustering architecture that achieves high throughput and low resource utilization across a wide set of algorithmic and system parameters

- Characterizing our system performance on a wide range of applications including image segmentation and big data analysis of real world datasets

The remainder of the paper is organized as follows. Section 7.2 presents more background of data clustering algorithms in general and the fundamental problems. Section 7.4 introduces our streaming data clustering algorithm. We explain our hardware design and optimization methods in Section 7.5, and show our experimental results in Section 7.6. We conclude in Section 7.7.

7.2 Data Cluatering

Data clustering is one of the fundamental problems in data analytics, pattern recognition, data compression, image analysis, and machine learning [53, 70, 60, 39]. Its goal is to group data objects that most resemble one another into the same cluster based upon some metric of similarity, or equivalently to separate data items that are relatively distinct into separate clusters.

Clustering algorithms can exhibit vastly different performance depending on the application, thus one must employ the algorithm that best matches the characteristics of the data set. For example, *k*-means is one of the oldest and simplest clustering algorithm. It partitions input observations into *k* groups, each of which is represented by a single mean point of the cluster. It is frequently used, likely due to its simplicity, but its basic assumption limits the separability of the data. Furthermore, it uses an iterative approach that does not scale well. There are many variations of the *k*-means algorithm e.g., [10, 11], and other algorithmic approaches, such as *BIRCH* or *DBSCAN* [71, 24] developed to provide better performance or work with datasets with different properties.

Increasing amounts of data are created in our daily life. These “big data” sets can be large, high-dimensional, diverse, variable, and delivered at high rates. More importantly, they are commonly time sensitive. The data must be analyzed quickly to extract actionable knowledge. In order to improve our ability to extract knowledge and insight from such complex and large data sets, we must develop efficient and scalable techniques to analyze these massive data sets being delivered at high rates.

Online data clustering algorithms handle unbounded streaming data without using a significant amount of storage. Thus, they provide a fast technique that maps well to hardware. However, online clustering has its drawbacks. Generally online algorithms look at the data only once. While this limits the storage, and thus allows for scalability and more efficient hardware implementations, it can reduce the accuracy compared to other iterative approaches that perform multiple passes over the data. For example, if the data characteristics evolve over time, the online algorithms can get stuck in a local optimum. These issues make it non-trivial to perform an accurate clustering using online algorithms. Yet these algorithms have good scalability and map efficiently into hardware.

We propose a multilevel, online data clustering method that is accurate while providing a scalable hardware architecture that is suitable for implementation in a heterogenous systems. Our method approximates multiple subclusters from streaming data first, then applies a problem specific clustering algorithm to these subclusters. Each subcluster is represented using a set of centroids which are estimated with different parameters independently. Each subcluster module accepts streaming input data and keeps updating the centroids set based upon the new data object. The next step to cluster these approximated points maps centroids to clusters, which is determined by the dataset properties. In our method, one cluster can have more than one center points unlike the *k*-means algorithm which has a single representative point per one cluster.

We carefully profile the algorithm and partition the workload across hardware and software. The subclustering process handles a massive amount of data and is a very demanding operation. Therefore we optimize its hardware implementation to perform a one-pass process while minimizing computation and space complexity. The next module deals with a relatively small set of data, so it can be processed either in software or hardware depending on a system goals.

7.3 Related Work

There are many clustering algorithm that target different data set properties. Generally it is up to the user to choose the “best” algorithm. Clustering algorithms can be largely divided into several groups, and, in this paper, we consider three popular clustering groups: partitioning, hierarchical, and density-based. We will focus on three algorithms – one from each group (*k*-means, BIRCH, and DBSCAN). And we specifically compare our work to existing hardware accelerated approaches.

k-means is the most used partitioning method, which is commonly known as Lloyd’s algorithm. It finds a set of centroids that represents data clusters. It is the simplest method that is frequently used in practical applications. There exist many variation of *k*-means algorithm, such as *k*-median, *k*-medoids, or *k*-means++. However, its inherent iterative solution for an optimal centroid set is highly compute and data intensive. As such, there have been many efforts to improve its computing performance [10, 11]. Hierarchical approaches build a hierarchy of clusters based on their similarity, and split down or merge up close clusters. The BIRCH algorithm is a well-known hierarchical algorithm [71]. It minimizes the number of processing passes and is capable of handling large datasets in a limited memory. DBSCAN algorithm is a density-based clustering method [24]. It scans dataset iteratively and finds a data group packed in high density. It can cluster an arbitrarily density shape dataset and has a notion of noise, which makes it robust to outliers.

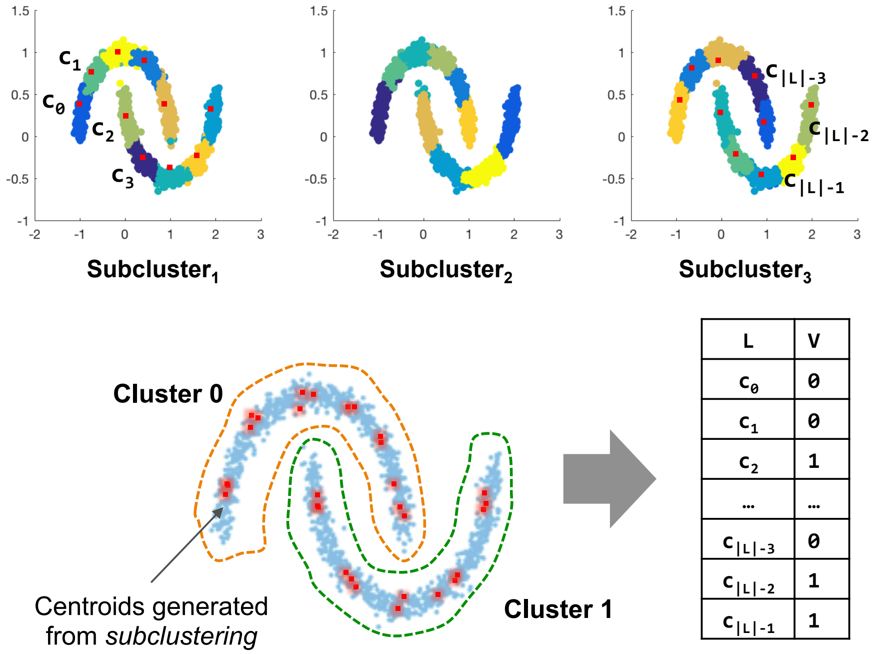
Each algorithm has limitations. The quality of k-means is highly dependent on the initial seed, and it is limited to clusters separable by d-dimensional spherical densities. Its objective function is sensitive to outliers, and its iterative operation makes it hard to scale. *BIRCH* uses a two-pass process to reduce these issues, but it is sensitive to parameters. And it uses a CF-tree data structure which is difficult to implement efficiently in hardware. *DBSCAN* is also very sensitive to parameters in terms of accuracy. This algorithm requires iterative operation and needs data to stay in a memory, which makes hard to map to hardware.

There are several projects aimed to accelerate clustering algorithms using a custom hardware or heterogeneous system. Hussain et al. [34, 33] accelerate k-means on an FPGA to perform gene analysis. They compare their FPGA implementation with a GPU implementation, and demonstrate speedup and improved energy efficiency on the FPGA. However, the on-chip memory capacity limits the size of data set to a small number of dimensions and a small number of centroids. Lin et al. [46] present a k-means hardware accelerator that uses a triangle inequality to reduce the computational complexity. The accelerator can handle 1024-dimensional data from an external DDR memory, but can only handle a small number (1024) of data points. More recently, Abdelrahman et al. [8] explores k-means on a shared memory processor-FPGA system. They partition the k-means workload across CPU and FPGA. They achieve 2.9× speed up against CPU only implementation and 1.9× faster than an accelerator alone design. However, their work does not support high dimensional data clustering and presents limited results for small numbers of clusters.

Some approaches merge hardware acceleration and data structure optimizations.

Chen et al. [18] implements a hierarchical binary tree on an FPGA. The tree is generated by splitting the data set recursively. Similarly, Winterstein et al. [64] use a kd-tree and with on-chip dynamic memory allocation in an attempt to efficiently use memory resources. While the accelerator traverses the tree, it updates a set of centroids. This process reduces the computational load, however, their design requires preprocessing to build a tree, and it does not handle a high dimensional data. In general, larger trees do not fit on an on-chip FPGA memory, and traversing the tree requires frequent irregular data accesses that limit performance. Our solution does not have these limitations.

As a demand for clustering big data analysis increases, streaming clustering algorithms have gotten more attention as they are more easily scaled to larger data sets. *StreamKM++*[10] uses a non-uniformly adaptive sampling approach for *k*-means to handle streaming data. It uses a *coreset tree* data structure to bound the data set size while streaming in data. Ailon et al. [11] suggests a streaming approximation of *k*-means by expanding *k*-means clustering algorithm in hierarchical manner. These streaming methods provide a good approximation of *k*-means and improves its performance by minimizing memory accesses. However, these methods still have significant computational complexity, which hinders their efficiency when mapped to hardware. For example, the *coreset tree* data structure used in *StreamKM++* is hard to implemented in hardware. And the approximation algorithm in [11] still has iteration within its process. Our method approximates input data into centroids more efficiently in a streaming way. We use vector quantization [47] to build a streaming clustering architecture on an FPGA. The approximation algorithm minimizes the computation and space complexity, which yields higher performance with less memory space needed. Our architecture is described in more detail the next section.



L	V
c_0	0
c_1	0
c_2	1
…	…
$c_{\|L\|-3}$	0
$c_{\|L\|-2}$	1
$c_{\|L\|-1}$	1

Figure 7.1: Our multilevel clustering algorithm in two stages. The first stage clusters the same set of data multiple times (three subcluster modules in this example) similar to k-means. It generates $|L| = l$ centroids representing l subclusters that is more than target clusters. Then, it clusters them using an existing clustering algorithm to find a look up table, $L \times V$, that maps L centroids to the target clusters V. Data points in subcluster c_0 are clustered to cluster 0.

7.4 Streaming Clustering

In this section, we introduce our streaming clustering algorithm that handles an unlimited amount of data while achieving high accuracy and suitable for a wide range of applications.

7.4.1 Multilevel clustering

Our clustering method sets *multiple* representations for each cluster (see in Figure 7.1). The algorithm is divided into two main stages. We call the first stage *subclustering*. In this stage, n input data are clustered into l subclusters ($k < l < n$) in a similar manner to the k-means algorithm. In the second stage, some of these centroids are grouped together into a larger cluster. We call this *reduction* stage. Each subcluster is generated from the same set of input data, but use different parameters.

For example, three subcluster modules in Figure 7.1 consider the same data and generate k center points from each. These center points sets compose the l centroids. These l centroids, $L = \{c_0, c_1, ..., c_{l-2}, c_{l-1}\}$, are clustered in *reduction* stage using a problem specific clustering algorithm. Clustering algorithms are sometimes very sensitive on choosing right parameters or initial seeding points. Our method can reduce the dependency on a particular parameter by using these different subclustering results. The final result is a single lookup table that maps a set of centroids, $\{c_0, c_1, ..., c_{l-2}, c_{l-1}\}$, and corresponding cluster ID, $\{0, 1\}$. We have the final result clustered 0 or 1 either. For example, based on this look up table, all data having c_0 for the nearest centroid are assigned cluster 0, and other data closer to c_2 are clustered to cluster 1.

7.4.2 Streaming subclustering

Subclustering and *reduction* are key operations in our method. *Subclustering* stage processes a large size input data and generates centroids. *Reducing* handles a smaller set of approximated centroids. *Subclustering* is very data intensive and computationally demanding process while *reduction* is much lighter. To minimize overall computation and space complexity for big data analysis, we focus making the *subclustering* operation into a hardware friendly streaming algorithm. It is based on a streaming version of vector quantization, which is also closely related competitive learning or a leader-follower clustering algorithm [22].

Vector quantization is used for data compression in signal processing. It partitions the data into subsets (clusters), which are modeled as probability density functions represented by a prototype vector (centroid). The simplest version for vector quantization picks data vector randomly from a given dataset. Then, it determines its appropriate

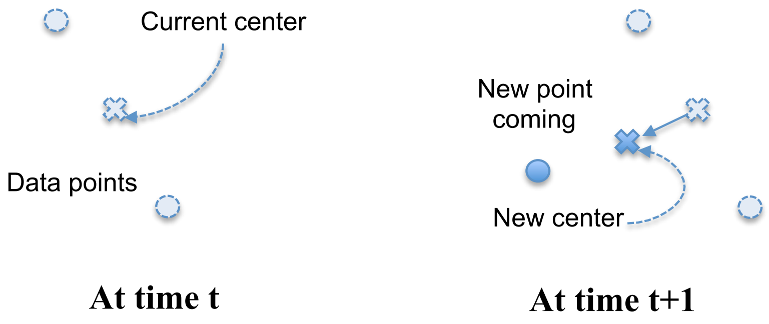


Figure 7.2: When a new data point comes in, a center point that locates close moves toward the new point. This process keeps updating and moving around this center point as a new data appears.

centroid, and updates the quantization vector centroid based upon that new data object. This vector moves to the current input points and it continues this process for the entire dataset. These steps can be done in one pass and easily implemented in a hardware architecture.

Our *subclustering* hardware module is built upon this streaming vector quantization technique. We assume that the input data is randomly ordered and stationary. Figure 7.2 shows an example of how our subclustering module works. If a new data point appears, the closest center point to the new data moves slightly towards it. It keeps updating and moving around this center point. Algorithm 1 presents the streaming subclustering algorithm. Input x is a d-dimensional streaming data point, and C is a current centroids of k clusters. The output is the new set of centroids, C. First, a processing core accepts input data, and it calculates distance from this current input to each centroid of k clusters. This point will be assigned to the closest cluster, and that cluster's center point is updated to consider the new input using the following equation:

$$c_{m_{t+1}} = (1-\alpha)\cdot c_{m_t} + \alpha \cdot x_t \tag{7.1}$$

The step size for this update is decided by the current input, the center point, and a learning rate, α. The learning rate is a weight of the current input data where x_t is a current input at time t, c_{m_t} is a clustered center point for x_{t-1}, and $c_{m_{t+1}}$ is an updated

```
Algorithm 1: Streaming subclustering $(x, C)$
    Input  : $x$ is a streaming input in $d$ dimension,
             $C$ is a current set of centroids
    Output : $C$ is the latest set of centroids
    Accept a new input $x$
    Calculate distance between each center point $c \in C$ and the current input $x$
    Get a center point of the nearest cluster, $c_m$.
    Move $c_m$ closer to $x$
    Return the current $C$
```

center point.

The initial seeding problem is an important issue for clustering algorithms, such as k-means or vector quantization, to find a global optimum. k-means++ defines the precondition problem in k-means and suggests a solution for better accuracy. In other works, initial centroids are randomly chosen in general. Our method accepts an unbounded input stream, so we can feed subclustering modules a random points or use a precalculated set from software side with a small subset of data in first part of data sequence using k-means.

7.4.3 Reducing

The reducing stage is defined at a high level in Equation (7.2). Its input is $K = \bigcup_{i=1}^{m} K_i$ such that $K_i = subcluster_i(input)$ where m is the number of subcluster modules; there are three subclusters in Figure 7.1, for example. The output is $L \times V$, a lookup table that maps centroids to assigned cluster *ID*s.

$$Reduction : K \to L \times V \qquad (7.2)$$

A *reduction* stage can use any clustering algorithm depending on applications or dataset properties. In this paper, we demonstrate our system with three clustering methods for this stage: *minimum cost pick*, *DBSCAN*, and *BIRCH*. *Minimum cost pick* is the simplest method. Each *subclustering* module calculates a cost, an averaged sum of distances between a centroid and data points within the cluster. It compares cost values from every *subclustering* modules and chooses a single set that has the minimum cost. In this case, $L = \{K_i\}$ and $V = \{1,2,...,|V|\}$ such that *DBSCAN* and *BIRCH* algorithms cluster these centroids as input. *DBSCAN* keeps scanning these points multiple times and finds associated data points within a fixed distance. The distance is defined as a parameter, $epsilon$, and if a cluster does not have enough number of elements, $minpts$, it considers the cluster as a noise. In this method, $L = K$ and $V = dbscan(K, epsilon, minpts)$. *BIRCH* generates a tree structure based on two different distance metrics while scanning input data, called CFtree. Then, it scans the initial CFtree and rebuilds a smaller one, and it applies a clustering algorithm to all the tree leaf entries. For *BIRCH* algorithm, $L = K$ and $V = birch(K, threshold)$.

7.4.4 Shuffling data

Our streaming subclustering module runs based on an assumption that the order of incoming data is random and stationary. However, it does not necessarily hold for all applications. Therefore we add the ability to randomize the dataset. In a streaming process, the processor does not have a control over input sequence coming that is unbounded. To make this practical, we shuffle a data array within a fixed window. This randomness makes our method more robust and improves accuracy in final results.

Randomization also helps the streaming approach better approximate a non-streaming algorithm. For example, *k*-means keeps revisiting input data until a solution converges into an optimal point. Instead of scanning the entire dataset multiple times, which is expensive in hardware, we divide the input dataset into several windows. The algorithm scans each window only once, which approximates scanning the original data iteratively. We can vary the size of the window. A larger shuffling window provides a result that closer to an offline method though it requires more hardware resources. Our experiment shows a fully sorted dataset results in a higher error, which can be significantly reduced through randomization to provide similar accuracy as *k*-means.

7.4.5 Design parameters

Data clustering is employed in all kind of different data sets that vary in dimension, the number of clusters, data size, data type, or other attributes. For example, multimedia data commonly has RGB 3-dimensional data, but other data can have significantly more features [44]. Our proposed system accommodates different clustering parameters for various applications.

We have several parameters to build a streaming *subclustering* core on a hardware: dimension d, the number of clusters k, and learning rate α. A streaming system does not have a limitation on data size. So the dimension and the number of clusters mainly determine throughput performance and resource utilization. Therefore, we focus on optimizing a hardware core to handle different dimensions and different numbers of clusters while retaining the maximum throughput. The learning rate α affects the updating centroids operation. We set different *subclustering* modules to run with different learning rates. Clustering algorithms in *reduction* stage also has important parameters, e.g. $epsilon$ and *minpts* for *DBSCAN*. However, they are highly application-specific and depend on data set properties, so we do not discuss them. We present our experimental results with different design parameters in Section 7.6.

7.5 System Implementation

In this section, we describe our CPU-FPGA heterogeneous system design. The input is an unbounded data stream, and output is a lookup table that describes the

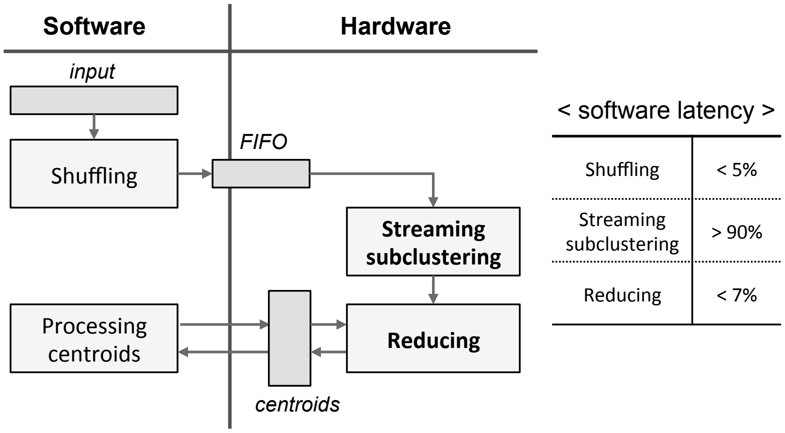


Figure 7.3: Overall system flow of our heterogeneous clustering system. *Streaming subclustering* is the most computationally intensive function, so it is accelerated in hardware. The *Reducing* function can be placed in hardware or software.

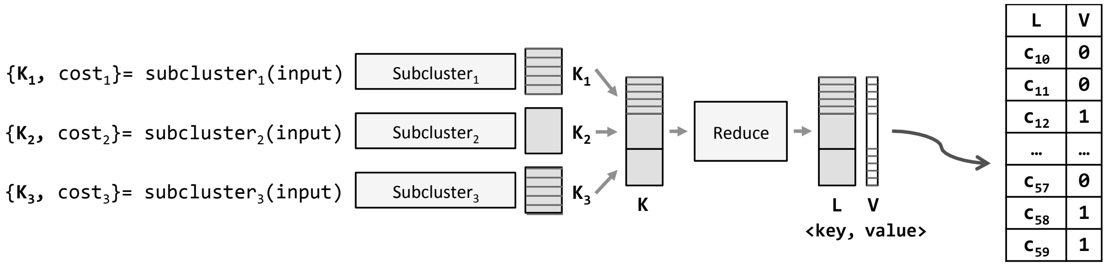


Figure 7.4: Hardware design for the multilevel streaming clustering. *Streaming subclustering* modules are fully parallelized since they are independent from each other. *Reducing* module merges subcluster centroids and finds final cluster ID for each point.

centroids and clusters.

7.5.1 Heterogeneous system

The overall system flow consists of *shuffling*, *streaming subclustering*, and *reduction* (see in Figure 7.3). According to our software profiling results using example datasets, *subclustering* stage takes almost 90% of total latency on average. *Shuffling* is less than 5%, and *reduction* is around 7%.

We focus on accelerating the main bottleneck module, *streaming subclustering* stage, and additionally implement *minimum cost pick* and *DBSCAN* methods in *reduction* stage on an FPGA. Figure 7.4 presents an accelerated core on an FPGA. *Shuffling* is implemented in software because it is not a compute intensive module, and its frequent data accesses limit it's acceleration capabilities on the FPGA. To communicate between CPU and FPGA, we employ RIFFA framework [36] and connect our FPGA core to RIFFA with the *AXIS* streaming interface.

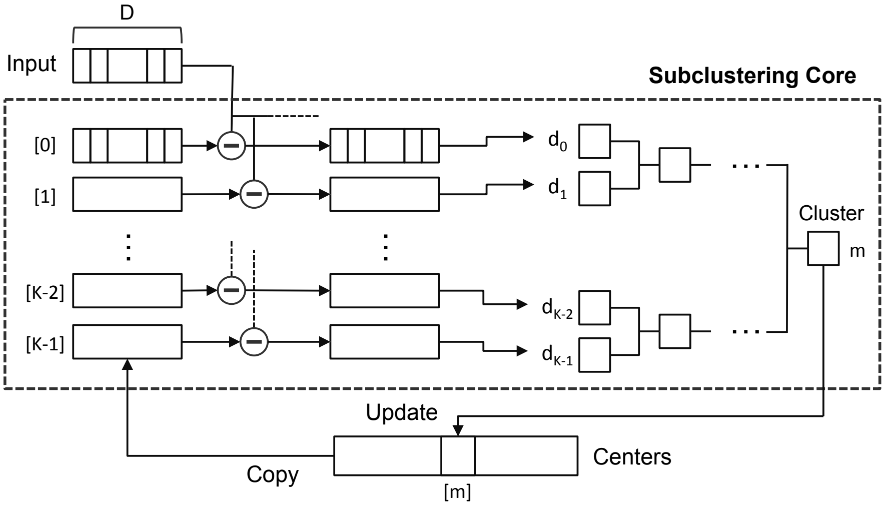


Figure 7.5: A processing core for streaming subclustering operation. It accepts *d*-dimensional inputs, decides on the appropriate cluster, and updates the corresponding centroid.

7.5.2 Subclustering module

The *Subclustering* module processes the same input sequence with different parameters multiple times. Each process is totally independent, so they are highly scalable in hardware. Our streaming approach minimizes computation complexity as well as hardware resources and we parallelize these independent operations.

Figure 7.5 presents the subclustering core. The accelerator core starts by calculating the distance between the current input data object and the centroid for each of the *k* clusters. We used L_1 norm (i.e., Manhattan distance) for our distance metric. This exposes significant instruction level parallelism as the calculation performs an absolute difference operation on the dimension of input data object and elements of the centroid vector, and then sums these differences. More precisely it performs a sum of absolute differences which maps in a very efficient and scalable manner to an FPGA. The distance calculation is done in a fully parallel manner. We perform complete memory partitioning on the centroid points, i.e., they are stored in registers that can all be accessed in one cycle to allow for high bandwidth accesses.

The entire core is parametrized. A user defines parameters, d, k, α, and data type of the data objects. A data clustering core is automatically synthesized based upon these parameters. The entire process is fully pipelined. Every time a new input arrives, the core continues processing and generates one output per input. It takes d clock cycles (dimension of the data objects) to accept d data objects. So the optimal pipeline initiation interval (II), i.e., our target performance, is d clock cycles.

7.5.3 Reducing module

We implement the *minimum cost pick* and *DBSCAN* methods on an FPGA for the *reduction* stage. The *BIRCH* algorithm uses a tree based data structure that is non-trivial to be implemented on hardware, so we leave that in software. *Minimum cost pick* simply compares cost values from every *subclustering* modules and chooses the one set that has the minimum cost value. This module is easily implementable in hardware. The *DBSCAN* algorithm scans the dataset multiple times. This iterative scanning operation causes high latency for a large size datasets and uses many resources. To achieve high performance, it requires intensive hardware optimization. However, since we handle much smaller size data in *reduction* module than in the *subclustering* module, it does not need high performance.

We utilize an open source code for DBSCAN [68] to synthesize a hardware architecture using a high level synthesis tool. We optimize the code to use a FIFO module to keep the associated candidate data point for a cluster, instead of a linked list data structure originally used in software.

7.6 Experimental Results

7.6.1 Test environment

We evaluated our proposed design on a CPU-FPGA heterogeneous system. Our test system has Intel i7 core 4 GHz and 16 GB DDR in software and a Xilinx Virtex 7 FPGA device, XC7VX485T-2FFG1761C, in hardware. We built an accelerator core using Xilinx Vivado HLS 2016.4. We integrated the FPGA core with RIFFA [36] to connect to a CPU and used the Vivado 2016.4 to generate a bitstream file.

We verify our approach using several different application datasets with different parameters. Table 7.1 presents eight example datasets: synthetic datasets of different shapes in 2D and 3D dimensions – *blobs, moons, circles,* and *3D clouds*, datasets from from UCI Machine Learning Repository (*spambase* and *census 1990*) [44], and image segmentation examples in biomedical research – *cell images in 1D and 9D dimensions* [42]. Note that *3D clouds* is a same synthetic dataset used in [64], which is open source. The 9-dimensional *cell images* data is generated by 3×3 convolutional windowing over 1-dimensional frame, and this convolutional segmentation method clusters the image based on its local variance in neighbor.

We apply different clustering algorithms in the *reduction* stage depending on the application. We use *DBSCAN* for blobs, moons, and circles dataset, *BIRCH* for image segmentations, and *minimum cost pick* for 3D clouds and high dimensional real world applications.

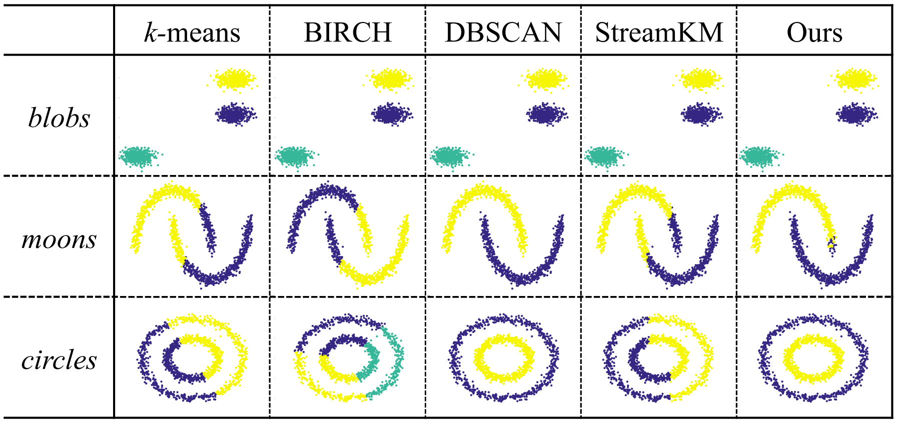
Table 7.1: Test datasets

data set	data size	dimension (d)	clusters (k)	datatype
blobs	1,500	2	3	float
moons	1,500	2	2	float
circles	1,500	2	2	float
3D clouds	16,384	3	128	int
spambase	4,601	57	10	int, float
census 1990	2,458,285	68	10	int
cell image (1D)	131,072	1	10	int
cell image (9D)	131,072	9	10	int

7.6.2 Accuracy

We compare our clustering results for the example datasets to other clustering algorithms: k-means, *BIRCH*, *DBSCAN*, and *streamKM++*. Table 7.2 presents the clustering results for 2-dimensional synthetic datasets. k-means and *streamKM++* methods group data points centered around a single center point for each cluster, so they cannot find true clusters in *moons* and *circles*. On the other hand, *DBSCAN* is good at clustering these datasets. We choose this algorithm for our *reduction* process, and it clusters these datasets correctly.

Table 7.2: 2D synthetic data clustering results. k-means, *BIRCH* and *streamKM++* hardly find right results for non-spherical density shape datasets. Our method clusters them correctly.



We compare clustering costs – the mean of distances between each centroid and

Table 7.3: Comparison of cost results

	Kmeans	StreamKM++	Ours
3D clouds	159.85	158.28	164.21
spambase	97.79	113.92	103.24
census 1990	37.36	37.47	37.41

data points in a cluster. The cost value is estimated from an objective function value in (7.3) as k-means algorithm does. x's are n input data, $\{X_1, X_2, ...X_k\}$ present k clusters, and each of them is represented using a *single* center point, c_i. The cost value is estimated from an objective function value in Equation (7.3) that we have to minimize. A cost value is available only for *minimum cost pick* method. Our clustering method shows comparable results to k-means or *streamKM++* in Table 7.3.

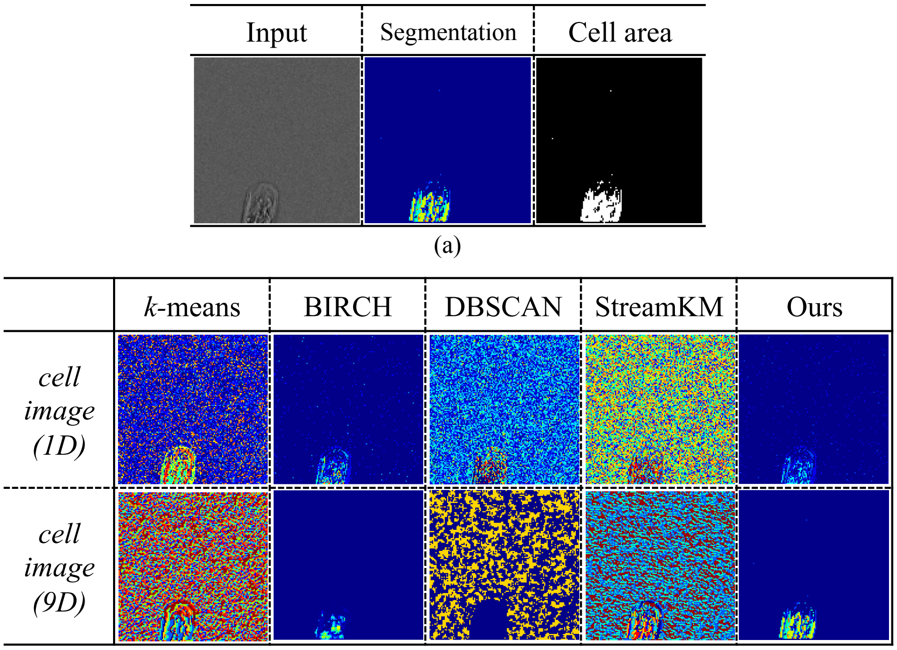
$$\underset{X}{\text{argmin}} \sum_{i=1}^{k} \sum_{x \in X_i} ||x - c_i||_L \tag{7.3}$$

We test our clustering method on image segmentation application. The segmentation results are presented in Table 7.4. Input image in this application is extremely noisy, and the image contrast is very low. Since the input is blurred in low intensity, it is nontrivial to separate particular pixel area and hard to achieve a good quality of segmentation results. *DBSCAN* hardly finds cell area since it is oversensitive to parameters. We use *BIRCH* algorithm in our *reduction* stage.

Data Shuffling

We observe that shuffled data gives a better approximation (close to k-means); the sorted data stream draws centroids off from the optimal locations. Figure 7.6 shows how data shuffling process changes the final cost value. *3D clouds* data is fully-sorted set with some initial clustering. Without shuffling, its streaming clustering results in a high cost value. We add the data shuffling module and increases the window size gradually. The cost value becomes lower and closer to k-means result. If the dataset is already

Table 7.4: Comparing segmentation results. (a) Input image is highly noisy and blurred in low contrast, so it is hard to achieve a good quality of segmentation result. (b) For example, *DBSCAN* does not recognize the difference between cell and background in a frame. Our method sees cell area clearly.



(a)

(b)

in random, it does not have much effect on the result, but if it is sorted, then shuffling operation is necessary. Thus sorting can be used depending upon the characteristics of the dataset.

7.6.3 Performance and resource utilization

FPGA Core Design

A generated hardware core is fully pipelined and runs in streaming manner. We set our target throughput as input bandwidth, which is determined by the data dimension *d* and the clock frequency. Each generated architecture can process data at line rate, i.e., one new datum per cycle.

Resource utilization increases almost linearly with respect to the data dimension

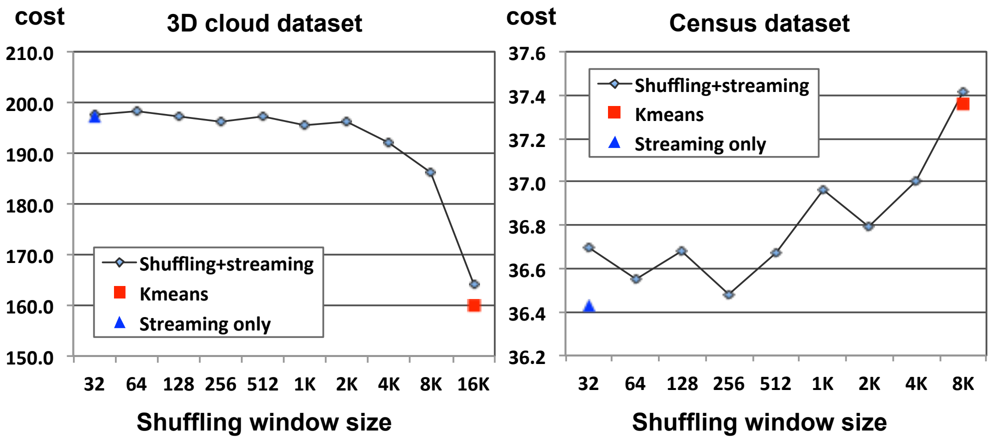


Figure 7.6: The cost values for different shuffling window sizes. Result becomes closer to *k*-means result with a larger shuffling window.

or the number of target clusters. We test our design with a maximum of 70 dimensional data. Targeting 10 clusters, it consumes 50.73% of BRAMs, 0 DSPs, 23.73% FFs, and 44.08% LUTs. To cluster 3-dimensional data into 128 clusters, it consumes 5.05% of BRAMs, 0 DSPs, 20.48% FFs, and 45.08% LUTs. We vary the learning rate at powers of two (e.g., $\alpha = 1/8$ through 1/64), which is synthesized to a right shift operation; thus the hardware module uses 0 DSPs. If we switch the parameter to an non power of tow, it consumes a few DSPs.

Table 7.5 compares our FPGA core performance results to other hardware accelerated works for *k*-means clustering algorithm [46, 64]. Our hardware core is highly optimized for pipelining and provides deterministic performance results decided by the data dimension d. It achieves more than 40 Msamples/s for 3-dimensional streaming data running at 125 MHz. It shows higher FPGA throughput than the results presented in [46, 64]. Considering the result in [64] does not include a latency from preprocessing, our clustering method outperforms their results, and can operate on unlimited size of data.

Table 7.5: FPGA core performance comparison with other FPGA implementations.

		Lin et al.	Winterstein et al.		Ours		
Data size (N)		1024	16384		Streaming		
Dimension (D)		1024	3		3	3	70
Clusters (K)		10	128		10	128	10
data type		8 bit unsigned int	16 bit unsigned int		16 bit unsigned int		
Max. capable data size		10000	65536		Infinite		
Throughput (Samples/s)		200 K	1.21 M (p = 1)	4.96M (p = 4)	45.93 M	41.83 M	1.83 M
Resources	LUTs	44194	-	14167	12785	133817	136872
	Registers	22521	-	24486	9156	14416	124383
	BRAMs	198	-	240	97	1045	104
	DSPs	-	-	186	0	0	0

Subclustering Module Analysis

The *Subclustering* stage is the most computationally intensive and data demanding module in our algorithm. We accelerate this module on an FPGA and evaluate our design with varying parameters: the dimension of data d, and the number of clusters k. It is based on a streaming approach, and performance and resource results do not depend on the dataset size. For the *subclustering* core analysis, we set a target clock frequency at $250MHz$ to evaluate its maximum performance.

Figure 7.7 and Figure 7.8 present the throughput and resource utilization results of a single *subclustering* core with different input data dimension size. We increase the dimension gradually from 1 up to 70. The number of clusters, k, is 16 in this experiment. The target throughput is determined by the input bandwidth, which is presented in Figure 7.7. High dimensional data needs more clock cycles to get input point, so input bandwidth is inversely proportional to its dimension. The processing core is able to achieve the target throughput in terms of clock cycles. It can produce output in every input, but the design complexity increases in higher dimensions. It results in running at a lower clock frequency, so the throughput result is less than the performance goal with higher dimensional data.

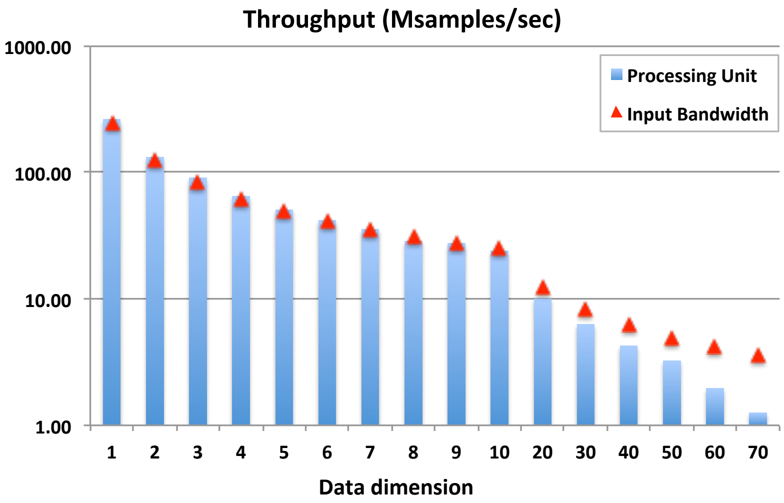


Figure 7.7: Throughput results by varying the data dimension. Input bandwidth is the maximum throughput that we can achieve, which depends on data dimension.

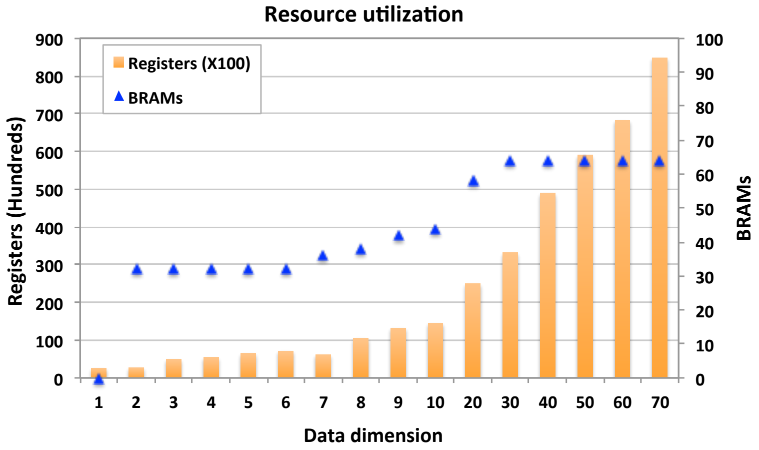


Figure 7.8: Resource utilization by varying the data dimension. Additionaly registers and BRAMs are required for the larger number of clusters k.

Figure 7.9 presents throughput and resource results by varying the number of clusters k. The data dimension in this experiment is fixed to 3. Ideally, the throughput result is determined by the data dimension, so the throughput result should be same. However, as k grows larger, the design complexity increases sharply and clock frequency gets lower. BRAMs used in the core module are partitioned completely. So the k value mostly decides BRAM usage, which is shown to be linear in Figure 7.9.

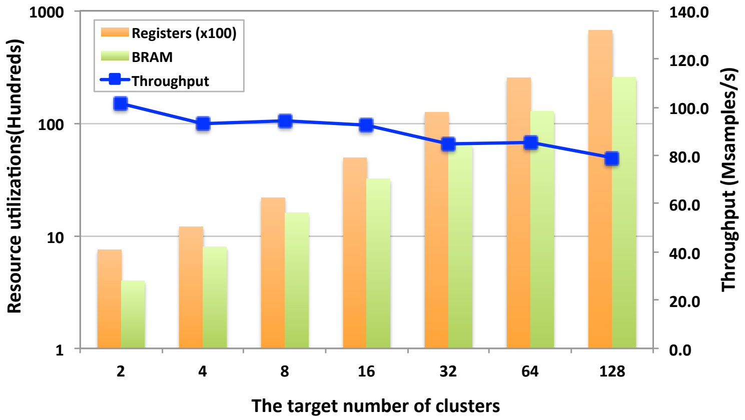


Figure 7.9: Throughput and resource utilization results by varying the number of clusters, *k*. The throughput result is mainly decided by the data dimension, but increasing complexity affects to clock period. Resource usage linearly increases according to *k*.

System performance

Table 7.6 presents overall system performance and resource utilization. The latency is measured for a window data, and the system throughput is based on the total latency. This includes data reading, which is the main bottleneck in the system performance. The data reading operation is basically a file I/O process to feed the system with a new data from external storage. It has much potential to be improved in software side, but we do not discuss an optimization as it is outside the scope of this paper. In spite of this software latency, our system performance is 1.39 Msamples/s for 3-dimensional data and much higher throughput up to 6.06 Msamples/s when ignoring the read latency.

We compare the system performance with *StreamKM++* which is one of state-of-art software approaches for large data set and presents the best throughput performance in software. For 9D cell image dataset, it runs 21× faster with our end-to-end system, and the core performance is up to 361× faster. For 68-dimensional *census 1990* dataset, the system performance results in 131×, faster and the core performance is 420× faster.

Table 7.6: System performance analysis and FPGA resource utilization. The reading module is a main bottleneck in the overall system, which includes file I/O for our test data.

		cell image	3d cloud	census 1990
Parameters	dimension (D)	9	3	68
	clusters (K)	10	128	10
	window size	16384	16384	8192
Latency (ms)	reading	13.76	10.66	28.49
	shuffling	0.99	0.63	3.82
	sending	5.01	1.82	13.57
	receiving	0.15	0.25	0.12
	total	19.94	13.42	465.03
Throughput (samples/s)	FPGA	13.89 M	41.67 M	1.84 M
	system	0.83 M (2.65 M)	1.39 M (6.06 M)	0.18 M (0.47 M)
Resources	LUTs	44389 (14.62%)	142593 (46.97%)	140027 (46.12%)
	FFs	58409 (9.62%)	156735 (25.81%)	134913 (22.22%)
	BRAMs	161 (8.0%)	1090 (52.91%)	149 (7.23%)

7.7 Conclusion

We develop a hardware oriented streaming clustering algorithm based on a multilevel clustering approach and its accelerated design on a CPU-FPGA heterogeneous system. Our clustering algorithm is able to process unbounded high dimensional streaming data while presenting comparable clustering results to existing algorithms. The proposed method approximates subclusters from a massive amount of data based using a streaming vector quantization, and then applies a problem specific clustering algorithm to these subclusters. We add an array shuffling module in the streaming process, which gives a better approximation to existing offline algorithms, such as *k*-means. We partition system workloads into a software and hardware to build a heterogenous hardware accelerated system. The experimental results show that our generated FPGA core processes more than 40 Msamples/s for 3-dimensional data and 1.78 Msamples/s for 70-dimensional data. The end-to-end system including all software processes achieves 1.39 Msamples for the same 3-dimensional dataset, which is 21× faster than a state-of-art software approach. Our hardware core is highly parameterized, so it can be easily extended for other applications.

This chapter, in full, has been submitted for publication of the material as it may appear in International Conference On Computer Aided Design (ICCAD), Lee, Dajung; Althoff, Alric; Richmond, Dustin; Kastner, Ryan, 2017 (accepted). There are small changes in format and phrasing as a chapter within this larger paper. The dissertation author was the primary investigator and author of this paper.

Chapter 8

Conclusion

Image based cell analysis system is promising for accurate cell analysis and its characterization. It is capable of providing sophisticated information for various cellular properties. However, extracting cellular features from low-resolution microscopic images is compute-intensive, and it commonly requires a high-throughput and low-latency solution for massive cell analysis, so building a real-time imaging flow cytometry system is a non-trivial problem. In this work, we designed a hardware accelerated system to achieve these performance goals while capable of capturing advanced cellular contents from high frame rate camera video. This paper describes a novel hardware accelerated system for imaging flow cytometry.

We introduced a hardware-friendly image analysis algorithm to extract cellular morphological features from microscopic images and its accelerated designs in different architectures: GPU and FPGA. We demonstrated we can achieve considerable performance improvement in both hardware platforms. However, we concluded that the GPU design is not a good fit for this type of applications because of the strict latency constraints and the inherent high latency bottleneck of GPU architecture. In an advanced morphological feature analysis system on an FPGA, we achieved the high-throughput and low-latency performance goals and demonstrated the system running with sets of real microscopic images accurately. We measured morphological information in our system, which can be used to estimate mechanical properties, such as cell circularity or deformability. Our work focused on extracting the external shape of cell, but this information can be used to sort cells in the post processing,

We extended our system to observe more complicated cell features or irregular cell shapes. It is based on image segmentation approach. We explored different image segmentation methods. They include luminance thresholding, iterative selection, hysteresis with temporal signature, *k*-means clustering, and convolutional window based methods. We carefully profiled these algorithms in terms of accuracy and efficiency in hardware design. Data clustering based segmentation algorithm is commonly used for such an applications and gave us one of the best results and the biggest potential for general applications.

We extended this segmentation approach into a general data clustering problem, mostly focusing on streaming data analysis. We designed a hardware oriented streaming data clustering approach based on a multilevel clustering approach. We accelerated our new approach on a CPU-FPGA heterogeneous system and suggested a parameterizable hardware designing method for various datasets. It is capable of handling unbounded high dimensional streaming data and presents comparable clustering results to existing algorithms. On a hardware accelerated system, we demonstrated our data clustering system outperforms other state-of-art clustering software and hardware approaches.

Bibliography

[1] Catapult high level synthesis, https://www.mentor.com/.

[2] The imagestreamx by amnis.

[3] Matlab fpga design and soc codesign, https://www.mathworks.com/solutions/fpga-design.html.

[4] Opencl fpga, https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[5] Phantomcamera, https://www.phantomhighspeed.com/products/phantom-camera-products.

[6] Synopsys synphony c compiler, https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/synphony-c-compiler.html.

[7] Vivado high-level synthesis, https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[8] Tarek S Abdelrahman. Accelerating k-means clustering on a tightly-coupled processor-fpga heterogeneous system. In *Application-specific Systems, Architectures and Processors (ASAP), 2016 IEEE 27th International Conference on*, pages 176–181. IEEE, 2016.

[9] Manouk Abkarian, Magalie Faivre, and Howard A Stone. High-speed microfluidic differential manometer for cellular-scale hydrodynamics. *Proceedings of the National Academy of Sciences of the United States of America*, 103(3):538–542, 2006.

[10] Marcel R Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. StreamKM++. *Journal of Experimental Algorithmics*, 17(1):2.1–30, July 2012.

[11] Nir Ailon, Ragesh Jaiswal, and Claire Monteleoni. Streaming k-means approximation. In *Advances in Neural Information Processing Systems*, pages 10–18, 2009.

[12] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[13] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C Ling, and Gordon R Chiu. An opencl (tm) deep learning accelerator on arria 10. *arXiv preprint arXiv:1701.03534*, 2017.

[14] Sugato Basu, Arindam Banerjee, and Raymond Mooney. Semi-supervised clustering by seeding. In *In Proceedings of 19th International Conference on Machine Learning (ICML-2002*. Citeseer, 2002.

[15] David G Buschke, Jayne M Squirrell, Hidayath Ansari, Michael A Smith, Curtis T Rueden, Justin C Williams, Gary E Lyons, Timothy J Kamp, Kevin W Eliceiri, and Brenda M Ogle. Multiphoton flow cytometry to assess intrinsic and extrinsic fluorescence in cellular aggregates: applications to stem cells. *Microscopy and Microanalysis*, 17(04):540–554, 2011.

[16] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[17] James Che, Victor Yu, Manjima Dhar, Corinne Renier, Melissa Matsumoto, Kyra Heirich, Edward B Garon, Jonathan Goldman, Jianyu Rao, George W Sledge, Mark D Pegram, Shruti Sheth, Stefanie Jeffrey, Rajan P Kulkarni, Elodie Sollier, and Dino Di Carlo. Classification of large circulating tumor cells isolated with ultra-high throughput microfluidic vortex technology. *Oncotarget*, 7(11):12748–12760, 2016.

[18] Tse-Wei Chen and Shao-Yi Chien. Flexible hardware architecture of hierarchical k-means clustering for large cluster number. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(8):1336–1345, 2011.

[19] Karen Cheung, Shady Gawad, and Philippe Renaud. Impedance spectroscopy flow cytometry: On-chip label-free cell differentiation. *Cytometry Part A*, 65(2):124–132, 2005.

[20] Srdjan Coric, Miriam Leeser, Eric Miller, and Marc Trepanier. Parallel-beam backprojection: an fpga implementation optimized for medical imaging. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 217–226. ACM, 2002.

[21] Alden A Dima, John T Elliott, James J Filliben, Michael Halter, Adele Peskin, Javier Bernal, Marcin Kociolek, Mary C Brady, Hai C Tang, and Anne L Plant. Comparison of segmentation algorithms for fluorescence microscopy images of cells. *Cytometry Part A*, 79(7):545–559, 2011.

[22] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[23] Jaideep S Dudani, Daniel R Gossett, TK Henry, and Dino Di Carlo. Pinched-flow hydrodynamic stretching of single-cells. *Lab on a Chip*, 13(18):3728–3734, 2013.

[24] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[25] Shady Gawad, Laurent Schild, and Ph Renaud. Micromachined impedance spectroscopy flow cytometer for cell analysis and particle sizing. *Lab on a Chip*, 1(1):76–82, 2001.

[26] Keisuke Goda, Ali Ayazi, Daniel R Gossett, Jagannath Sadasivam, Cejo K Lonappan, Elodie Sollier, Ali M Fard, Soojung Claire Hur, Jost Adam, Coleman Murray, Chao Wang, Nora Brackbill, Dino Di Carlo, and Bahram Jalali. High-throughput single-microparticle imaging flow analyzer. *Proceedings of the National Academy of Sciences*, 109(29):11630–11635, 2012.

[27] Keisuke Goda, Dino Di Carlo, and Bahram Jalali. Ultrafast automated image cytometry for cancer detection. In *Engineering in Medicine and Biology Society (EMBC), 2013 35th Annual International Conference of the IEEE*, pages 129–132. IEEE, 2013.

[28] Daniel R Gossett, TK Henry, Serena A Lee, Yong Ying, Anne G Lindgren, Otto O Yang, Jianyu Rao, Amander T Clark, and Dino Di Carlo. Hydrodynamic stretching of single cells for large population mechanical phenotyping. *Proceedings of the National Academy of Sciences*, 109(20):7630–7635, 2012.

[29] Hernán E Grecco, Sarah Imtiaz, and Eli Zamir. Multiplexed imaging of intracellular protein networks. *Cytometry Part A*, 2016.

[30] Pierre Greisen, Simon Heinzle, Markus Gross, and Andreas P Burg. An fpga-based processing pipeline for high-definition stereo video. *EURASIP Journal on Image and Video Processing*, 2011(1):1–13, 2011.

[31] Jochen Guck, Stefan Schinkinger, Bryan Lincoln, Falk Wottawah, Susanne Ebert, Maren Romeyke, Dominik Lenz, Harold M Erickson, Revathi Ananthakrishnan, Daniel Mitchell, Josef Käs, Sydney Ulvick, and Curt Bilby. Optical deformability as an inherent cell marker for testing malignant transformation and metastatic competence. *Biophysical journal*, 88(5):3689–3698, 2005.

[32] J El Hobbie, R Jasper Daley, and STTI977 Jasper. Use of nuclepore filters for counting bacteria by fluorescence microscopy. *Applied and environmental microbiology*, 33(5):1225–1228, 1977.

[33] Hanaa M Hussain, Khaled Benkrid, Ahmet T Erdogan, and Huseyin Seker. Highly parameterized k-means clustering on fpgas: Comparative results with gpps and gpus. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 475–480. IEEE, 2011.

[34] Hanaa M Hussain, Khaled Benkrid, Huseyin Seker, and Ahmet T Erdogan. Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 248–255. IEEE, 2011.

[35] Idaku Ishii, Tetsuro Tatebe, Qingyi Gu, Yuta Moriue, Takeshi Takaki, and Kenji Tajima. 2000 fps real-time vision system with high-frame-rate video recording. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 1536–1541. IEEE, 2010.

[36] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(4):22, 2015.

[37] Seunghun Jin, Junguk Cho, Xuan Dai Pham, Kyoung Mu Lee, Sung-Kee Park, Munsang Kim, and Jae Wook Jeon. Fpga design and implementation of a real-time stereo vision system. *IEEE transactions on circuits and systems for video technology*, 20(1):15–26, 2010.

[38] Shingo Kagami, Shoichiro Saito, Takashi Komuro, and Masatoshi Ishikawa. A networked high-speed vision system for 1,000-fps visual feature communication. In *2007 First ACM/IEEE International Conference on Distributed Smart Cameras*, pages 95–100. IEEE, 2007.

[39] Mehmet Koyuturk, Ananth Grama, and Naren Ramakrishnan. Compression, clustering, and pattern discovery in very high-dimensional discrete-attribute data sets. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):447–461, 2005.

[40] Sanjay Kumar and Valerie M Weaver. Mechanics, malignancy, and metastasis: the force journey of a tumor cell. *Cancer and Metastasis Reviews*, 28(1-2):113–127, 2009.

[41] Dajung Lee, Janarbek Matai, Brad Weals, and Ryan Kastner. High throughput channel tracking for jtrs wireless channel emulation. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4. IEEE, 2014.

[42] Dajung Lee, Pingfan Meng, Matthew Jacobsen, Hayson Tse, Dino Di Carlo, and Ryan Kastner. A hardware accelerated approach for imaging flow cytometry. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8. IEEE, 2013.

[43] Dajung Lee, Roger Moussalli, Sameh Asaad, and Mudhakar Srivatsa. Spatial predicates evaluation in the geohash domain using reconfigurable hardware. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, pages 176–183. IEEE, 2016.

[44] M. Lichman. UCI machine learning repository, 2013.

[45] Yen-Heng Lin and Gwo-Bin Lee. Optically induced flow cytometry for continuous microparticle counting and sorting. *Biosensors and Bioelectronics*, 24(4):572–578, 2008.

[46] Zhongduo Lin, Charles Lo, and Paul Chow. K-means implementation on fpga for high-dimensional data using triangle inequality. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 437–442. IEEE, 2012.

[47] Yoseph Linde, Andres Buzo, and Robert Gray. An algorithm for vector quantizer design. *IEEE Transactions on communications*, 28(1):84–95, 1980.

[48] Janarbek Matai, Ali Irturk, and Ryan Kastner. Design and implementation of an fpga-based real-time face recognition system. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 97–100. IEEE, 2011.

[49] Janarbek Matai, Dajung Lee, Alric Althoff, and Ryan Kastner. Composable, parameterizable templates for high level synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2016.

[50] Janarbek Matai, Pingfan Meng, Lingjuan Wu, Brad Weals, and Ryan Kastner. Designing a hardware in the loop wireless digital channel emulator for software defined radio. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 206–214. IEEE, 2012.

[51] Janarbek Matai, Dustin Richmond, Dajung Lee, Zac Blair, Qiongzhi Wu, Amin Abazari, and Ryan Kastner. Resolve: Generation of high-performance sorting architectures from high-level synthesis. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 195–204. ACM, 2016.

[52] Janarbek Matai, Dustin Richmond, Dajung Lee, and Ryan Kastner. Enabling fpgas for the masses. *arXiv preprint arXiv:1408.5870*, 2014.

[53] HP Ng, SH Ong, KWC Foong, PS Goh, and WL Nowinski. Medical image segmentation using k-means clustering and improved watershed algorithm. In *2006 IEEE Southwest Symposium on Image Analysis and Interpretation*, pages 61–65. IEEE, 2006.

[54] Cinzia Nobile, Dominika Rudnicka, Milena Hasan, Nathalie Aulner, Françoise Porrot, Christophe Machu, Olivier Renaud, Marie-Christine Prévost, Claire Hivroz, Olivier Schwartz, and Nathalie Sol-Foulon. Hiv-1 nef inhibits ruffles, induces filopodia, and modulates migration of infected lymphocytes. *Journal of virology*, 84(5):2282–2293, 2010.

[55] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Hormati. Amir, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Smith. Aaron, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[56] Ingrid Schmid, Wanda J Krall, Christel H Uittenbogaart, Jonathan Braun, and Janis V Giorgi. Dead cell discrimination with 7-amino-actinomcin d in combination with dual color immunofluorescence in single laser flow cytometry. *Cytometry*, 13(2):204–208, 1992.

[57] Ingrid Schmid, Christel H Uittenbogaart, Birgitte Keld, and Janis V Giorgi. A rapid method for measuring apoptosis and dual-color immunofluorescence by single laser flow cytometry. *Journal of immunological methods*, 170(2):145–157, 1994.

[58] Roger A Schultz, Thomas Nielsen, Jeff R Zavaleta, Raynal Ruch, Robert Wyatt, and Harold R Garner. Hyperspectral imaging: a novel approach for microscopic analysis. *Cytometry*, 43(4):239–247, 2001.

[59] Neeraj Sharma and Lalit M Aggarwal. Automated medical image segmentation techniques. *Journal of medical physics*, 35(1):3, 2010.

[60] Andriy Shepitsen, Jonathan Gemmell, Bamshad Mobasher, and Robin Burke. Personalized recommendation in social tagging systems using hierarchical clustering. In *Proceedings of the 2008 ACM conference on Recommender systems*, pages 259–266. ACM, 2008.

[61] Vinay Swaminathan, Karthikeyan Mythreye, E Tim O'Brien, Andrew Berchuck, Gerard C Blobe, and Richard Superfine. Mechanical stiffness grades metastatic potential in patient tumor cells and in cancer cell lines. *Cancer research*, 71(15):5075–5080, 2011.

[62] Henry Tat Kwong Tse, Pingfan Meng, Daniel R Gossett, Ali Irturk, Ryan Kastner, and Dino Di Carlo. Strategies for implementing hardware-assisted high-throughput cellular image analysis. *Journal of the Association for Laboratory Automation*, 16(6):422–430, 2011.

[63] RY Tsien, TJ Rink, and M Poenie. Measurement of cytosolic free ca 2+ in individual small cells using fluorescence microscopy with dual excitation wavelengths. *Cell calcium*, 6(1):145–157, 1985.

[64] Felix Winterstein, Samuel Bayliss, and George A Constantinides. FPGA-based k-means clustering using tree-based data structure. In *2013 23rd International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, June 2013.

[65] Zhenyu Wu and Richard Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 15(11):1101–1113, 1993.

[66] Jimmy Xu, Nikhil Subramanian, Adam Alessio, and Scott Hauck. Impulse c vs. vhdl for accelerating tomographic reconstruction. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 171–174. IEEE, 2010.

[67] Feng Xue, Alex B Lennon, Katey K McKayed, Veronica A Campbell, and Patrick J Prendergast. Effect of membrane stiffness and cytoskeletal element density on mechanical stimuli within cells: an analysis of the consequences of ageing in cells. *Computer methods in biomechanics and biomedical engineering*, 18(5):468–476, 2015.

[68] Gagarin Yaikhom. Implementing the dbscan clustering algorithm, 2015.

[69] Minerva Yeung, Boon-Lock Yeo, and Bede Liu. Segmentation of video by clustering and graph analysis. *Computer vision and image understanding*, 71(1):94–109, 1998.

[70] Shihua Zhang, Rui-Sheng Wang, and Xiang-Sun Zhang. Identification of overlapping community structure in complex networks using fuzzy c-means clustering. *Physica A: Statistical Mechanics and its Applications*, 374(1):483–490, 2007.

[71] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, 1(2):141–182, 1997.

[72] Yi Zheng, John Nguyen, Yuan Wei, and Yu Sun. Recent advances in microfluidic techniques for single-cell biophysical characterization. *Lab on a Chip*, 13(13):2464–2483, 2013.