3-23-2018

Acceleration of k-Nearest Neighbor and SRAD Algorithms Using Intel FPGA SDK for OpenCL

Liyuan Liu
University of Windsor

Acceleration of k-Nearest Neighbor and SRAD Algorithms Using Intel FPGA SDK for OpenCL

By

Liyuan Liu

A Thesis
Submitted to the Faculty of Graduate Studies
through the Department of Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science
at the University of Windsor

Windsor, Ontario, Canada

2018

Acceleration of k-Nearest Neighbor and SRAD Algorithms Using Intel FPGA SDK for OpenCL

By

Liyuan Liu

APPROVED BY:

N. Zamani
Department of Mechanical, Automotive & Materials Engineering

H. Wu
Department of Electrical and Computer Engineering

M. Khalid, Advisor
Department of Electrical and Computer Engineering

March 20, 2018

Declaration of Previous Publication

This thesis includes 1 original paper that is in preparation to be submitted for publication, as follows:

Thesis Chapter	Publication title/full citation	Publication status*
Chapter 3	L. Liu and M. Khalid, “Acceleration of k-Nearest Neighbor Algorithm on FPGA using Intel SDK for OpenCL”	*In preparation*

I certify that I am the copyright owner(s) to the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as a graduate student at the University of Windsor.

I declare that, to the best of my knowledge, my thesis does not infringe upon anyone’s copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Field Programmable Gate Arrays (FPGAs) have been widely used for accelerating machine learning algorithms. However, the high design cost and time for implementing FPGA-based accelerators using traditional HDL-based design methodologies has discouraged users from designing FPGA-based accelerators. In recent years, a new CAD tool called Intel FPGA SDK for OpenCL (IFSO) allowed fast and efficient design of FPGA-based hardware accelerators from high level specification such as OpenCL. Even software engineers with basic hardware design knowledge could design FPGA-based accelerators.

In this thesis, IFSO has been used to explore acceleration of k-Nearest-Neighbour (kNN) algorithm and Speckle Reducing Anisotropic Diffusion (SRAD) simulation using FPGAs. kNN is a popular algorithm used in machine learning. Bitonic sorting and radix sorting algorithms were used in the kNN algorithm to check if these provide any performance improvements. Acceleration of SRAD simulation was also explored. The experimental results obtained for these algorithms from FPGA-based acceleration were compared with the state of the art CPU implementation. The optimized algorithms were implemented on two different FPGAs (Intel Stratix A7 and Intel Arria 10 GX). Experimental results show that the FPGA-based accelerators provided similar or better execution time (up to 80X) and better power efficiency (75% reduction in power consumption) than traditional platforms such as a workstation based on two Intel Xeon processors E5-2620 Series (each with 6 cores and running at 2.4 GHz).

Acknowledgements

First, I would like to show my greatest respect to my supervisor Dr. Mohammed Khalid for his patience on my study and research in the past two years at the University of Windsor. His guidance and encouragement helped me through one after another problem in my study. It is really my great honor work under his supervision.

I would like to thank my thesis committee members Dr. Huapeng Wu and Dr. Nader Zamani for offering me their precious time and giving me their encouragement and advice for my thesis.

I would like to acknowledge Ryan Tang and Huyuan Li for their technical assistance and suggestions on selecting my research topics. I would like to thank Mohammed Momen for discussion on experimental details.

I would like to thank my colleagues and friends, Pengzhao Song, Chunyu Mao, Qiang Zeng, Joe Lang, Zhe Zhang, Yangfan Hou, Jintao Han, and Terry Tian for their generous support and help.

Most importantly, I would like to thank my parents who gave me their infinite love and patience since I was born. I appreciate my wife for her endless assistance and inspiration in work and life. Without them, this thesis would not be completed.

Table of Contents

List of Tables

List of Figures

List of Acronyms

AI:	Artificial Intelligence
ASIC:	Application Specific Integrated Circuits
CAD:	Computer-aided Design
CLB:	Configurable Logic Block
CPE:	Custom Processing Element
CPU:	Central Processing Unit
FFT:	Fast Fourier Transform
FPGA:	Field Programmable Gate Array
GPU:	Graphic Processing Unit
HCS:	Heterogeneous Computing system
HDL:	Hardware Description Language
HLS:	High level Synthesis
HPC:	High Performance Computing
IFSO:	Intel FPGA SDK for OpenCL
IC:	Integrated Circuit
kNN:	k-Nearest-Neighbor
OpenCL:	Open Computing Language

RTL:	Register-transfer Level
SoC:	System on Chip
SRAD:	Speckle Reducing Anisotropic Diffusion
VHDL:	VHSIC Hardware Description Language

Chapter 1

Introduction

1.1 Motivation

According to IBM, human beings produce 2.5 quintillion (25 billion billion) bytes of data every day from their routine daily activities [1]. That is why people would like to call the current times as the age of “big data”. When talking about big data, developers mean the ability to acquire this data, analyze it and draw useful conclusions from it. These tasks are very computationally intensive and expensive. Hence there is lots of interest in using hardware accelerators for these tasks.

With the increasing popularity of data classification recently, Field Programmable Gate Arrays (FPGAs) as the parallel implementation platforms are suitable for many classification algorithms [2]. “Heterogeneous Computing system (HCS) consists of multiple computational cores with different configurations, connected by a high-speed LAN, for increased computational power and resources” [3]. HCS provides the best architecture for programmers to execute computation intensive problems by optimizing the HCS architecture for the given task. Currently, there is an upsurge in the computer design community experimenting with building HCS [4]. Figure 1 shows basic architecture of HCS.

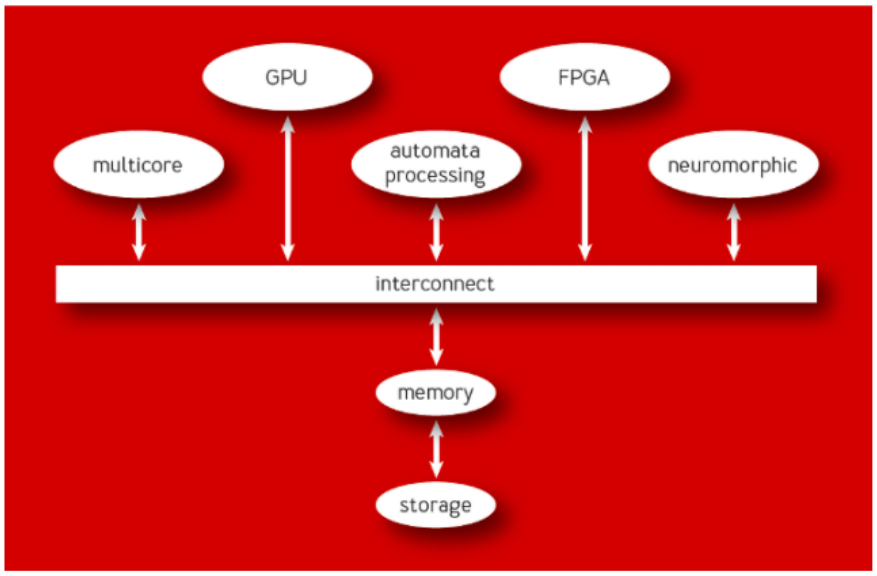


Figure 1. Basic architecture of HCS [5]

When accelerating classification algorithms on HCS, the difficulty is to manage the resource and increase the efficiency. Unlike CPU and GPU, FPGAs have the unique structure to take the advantages of CPU's complicated calculation ability and GPU's high speed parallel computing ability.

The Intel FPGA SDK for OpenCL (Open Computing Language) [6] aims to reduce the programming difficulty, run time and cost on parallel computing on FPGA. When using OpenCL, the knowledge of hardware design language such as Verilog or VHDL is not required from programmers, since the Intel FPGA SDK for OpenCL can synthesize high level language specification in OpenCL to generate optimized hardware accelerators for FPGAs. Intel FPGA SDK for OpenCL includes an offline complier for synthesizing OpenCL kernel code to Verilog descriptions and generating Quartus II compilation scripts for FPGA synthesis. As with other OpenCL platforms, the input specification consists of two parts: the host code and the kernel code. The host represents one processor coordinating execution and one or more processors executing OpenCL code used by programmers when writing specific functions (called kernels) that run on FPGA devices.

OpenCL can be widely used for accelerating computation intensive problems in different applications. One of most popular techniques utilized in dealing with data detection especially the tasks with artificial intelligence is machine learning. “Machine learning is a field of computer science that gives computers the ability to learn without being explicitly programmed” [7]. Nowadays, machine learning is used in various fields of data processing, such as image and video classification, document processing, speech recognition, etc. With the rise of heterogeneous computing systems, machine learning started to be implemented on such systems because of the computation intensive tasks required parallel hardware for fast execution. GPU platforms are quite suitable for running machine learning algorithms. However, power consumption is an issue for GPU requires expensive cooling systems to handle excessive heat dissipation. At the same time, some parts of algorithms can be only implemented on CPU since their structures are hard to parallelize and data transmission between CPU and GPU would cost computation time overhead.

Machine learning algorithms suitable for parallel structures can be implemented on FPGAs which provide fine grain programmable hardware resources with much lower power consumption compared to GPUs and multi-core CPUs. IFSO can be used to accelerate machine learning algorithms using FPGAs. The input OpenCL specification consists of host code and kernels. The host code can run embedded ARM processor on FPGA and kernels can be accelerated using programmable hardware in FPGA. The power consumption of embedded ARM processor is lower than traditional CPUs and GPUs. Therefore, machine learning algorithms implemented on embedded CPU and FPGA based heterogeneous computing systems has lower power consumption than CPU-GPU platforms.

However, IFSO has some limitations as well. The developers need to optimize the input OpenCL specification to obtain shorter execution time. The dataflow in the algorithm needs to be adjusted and memory needs to be managed well to minimize the execution time. More details on these issues will be provided later in chapter 3 and 4. Another limitation is that in specific complex algorithms, floating point data types are used. FPGA processing efficiency for these data types is not as good compared to GPUS and CPUs. The implementation on Intel FPGA SDK for OpenCL platforms has limitation on floating point values. No matter how well optimized the code is, computing floating point values on Intel SDK for OpenCL would cost more time than the implementation on CPU. Wherever possible, alternate data types such as fixed point should be used to obtain better performance without significantly degrading the accuracy of results.

1.2 Thesis Objectives

In this research, kNN algorithm enhanced with Bitonic sort and Radix sort algorithms and SRAD have been implemented using IFSO on FPGAs. The research goals were as follows:

- Implement optimized kNN algorithm (C++ code) using bitonic sort on multi-core CPU.
- Modify C++ code and create optimized OpenCL specification for kNN algorithm using bitonic sort.
- Implement optimized kNN algorithm (C++ code) using radix sort on multi-core CPU.
- Modify C++ code and create optimized OpenCL specification for kNN algorithm using radix sort.

- Implement SRAD simulation on multi-core CPU and create optimized OpenCL specification for SRAD simulation.
- Compare the results obtained using multi-core CPU and FPGA using performance and power consumption metrics.

1.3 Thesis Organization

The organization of this thesis is as follows: Chapter 2 introduces heterogeneous computing systems including FPGA architecture and accelerator hardware, high level synthesis, Intel SDK for OpenCL and machine learning.

In Chapter 3, kNN algorithm using bitonic sort was first implemented on multi-core CPU and then FPGAs using IFSO. The experimental results obtained from these platforms were compared and analyzed. Chapter 4 describes the acceleration of kNN algorithm using radix sort. The experimental results obtained from these platforms were compared and analyzed. In Chapter 5, SRAD was introduced and implemented on multi-core CPU. Then SRAD was implemented on FPGAs using IFSO. The results obtained from these platforms were compared and analyzed. Chapter 6 provides conclusion of this thesis and discussion of future work.

Chapter 2

Heterogeneous Computing System Overview

2.1 Heterogeneous Computing

In today's computing environment multiple processing platforms such as central processing units (CPUs), digital signal processors, reconfigurable hardware (FGPAs), and graphic processing units (GPUs) [8] are used. Such systems are called heterogeneous computing systems (HCS). They allow developers to select the best platform and architecture from many kinds of target platforms available to execute the task or optimize the task for the best efficiency. Currently HCS are becoming popular in many real world applications.

Applications built from HCS have specific goals to process workload behaviors (e.g., searching, sorting, and parsing), ranging from control intensive to data intensive (e.g., image processing, simulation and modeling, and data mining). HCS can also solve the problems like iterative methods, numerical methods, and financial modeling. In such areas, how to increase the computational efficiency from the underlying hardware is the main goal for developers. There is no perfect single architecture circuit to execute all kinds of applications, and the best way to execute is to select multiple platforms and combine them to a mixed architecture for best results.

The first discussion of software composition at the architectural level was first presented by Garlan and Shaw describing some architectural styles [9]. They claimed that "most systems typically involve some combination of several styles." In these systems, the combination architectures are HCS. However, in this paper the authors just stated an immature concept about HCS. They emphasized more attention on pure architecture styles. There are twelve software architecture styles mentioned in this paper:

- Layered
- Distributed processes and threads
- Pipes and filters
- Object-oriented
- Main program/subroutines
- Repositories
- Event-based (implicit invocation)
- Rule-based
- State transition based
- Process control (feedback)
- Domain-specific
- Heterogeneous

When building HCS, developers are required to precisely articulate the architecture of the systems they want to design, and most successful applications rely on more than one architecture style [10]. In one HCS, a variety architecture styles can be used. However, not all styles play the same important roles in the system. Some specific architecture can be implemented to execute most parts of problems and other architectures in the system help the main architecture to make up for the deficiencies in processing the other parts.

For example, Figure 2 shows the problem of designing a satellite ground system station system. The ground station takes the responsibility for receiving the information from satellite, processing the information, and sending the useful information to users.

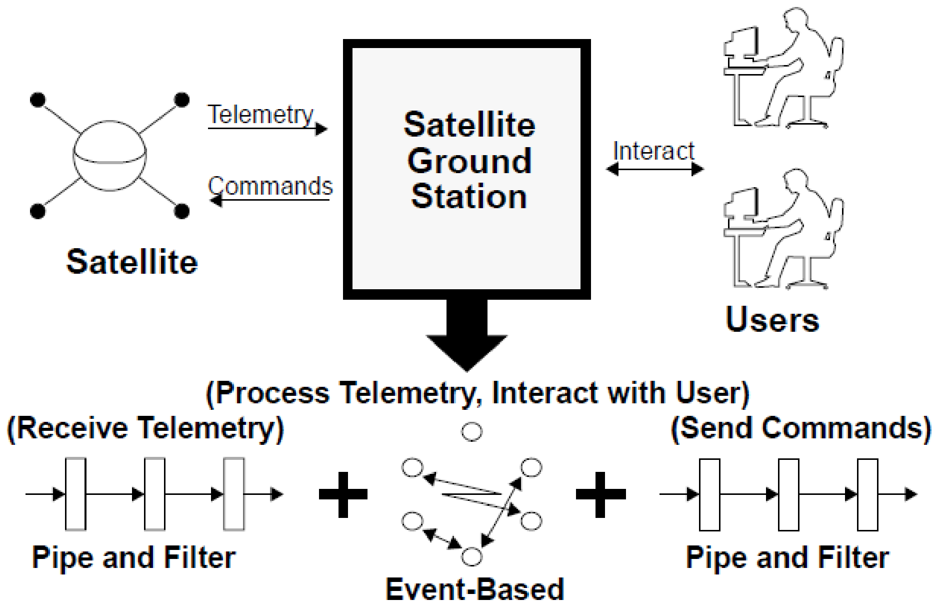


Figure 2. A Satellite Ground Station Using Multiple Architectural Styles [10]

This design consists of three main parts. The satellite is the first part to collect the telemetry from space and get commands from ground station. Satellite ground station is the second and main part to process telemetry and interact with the users. The Users part extracts the most useful information from the data sent by the satellite to the ground station. These multiple architectural styles system is clear and easy to understand representative example of HCS.

Another example of HCS is the desktop or laptop computer used these days. There are two main data processing units in each computer, which are CPU and GPU. CPU has multiple cores and is responsible for executing very complex control oriented tasks. The number of cores in a multi-core CPU are limited due to costs and efficiency (communication overhead) reasons. However, GPU has thousands of cores and is able to process large quantity of data in parallel. Figure 3 shows the structure of CPU and GPU.

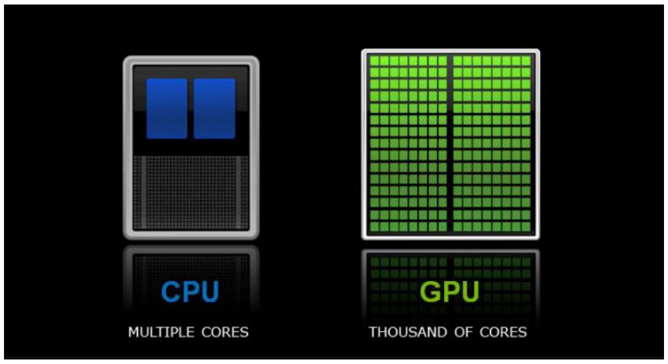


Figure 3. Structure of CPU and GPU [11]

In current computers CPU mainly controls the utilization of the whole system. GPU is responsible for heavy duty data processing (e.g. image processing).

2.2 FPGA Architecture and Accelerator Hardware

2.2.1 FPGA Architecture

Field Programmable Gate Arrays (FPGAs) were first designed almost two and a half decades ago [12]. An FPGA comprises of an array of programmable logic blocks. Blocks are connected to each other through programmable interconnect network. The advances in integrated circuit (IC) process technology make FPGAs a viable implementation alternative for large and complex digital designs. Hardware description language (HDL) models of digital designs can be optimized and synthesized for FPGAs [13] [14]. The main steps involved in synthesis are: logic synthesis, technology mapping, placement, routing and FPGA bit stream generation. The programmable structure of FPGAs makes a very flexible implementation medium for digital designs. They also consume much less power compared to high end CPUs and GPUs used in high performance computing (HPC).

FPGAs can be electrically programmed by developers in the field to build a variety of digital circuits for different real world applications. FPGAs provide large quantity of programmable hardware resources for lower cost (for low and medium product volumes) and faster time to market compared to Application Specific Integrated Circuits (ASIC). ASICs also require high non-recurring engineering (NRE) costs.

FPGAs consist of:

- Programmable logic blocks which implement logic functions.
- Programmable routing that connects these logic functions.
- I/O blocks that are connected to logic blocks through routing interconnect and that make off-chip connections [12].

Figure 4 describes the overview of FPGA architecture. Configurable logic blocks (CLBs) are laid out in a two dimensional grid and are interconnected by programmable routing resources. I/O blocks are also interconnected to programmable routing. These structures enable the FPGA to provide programmable logic and routing.

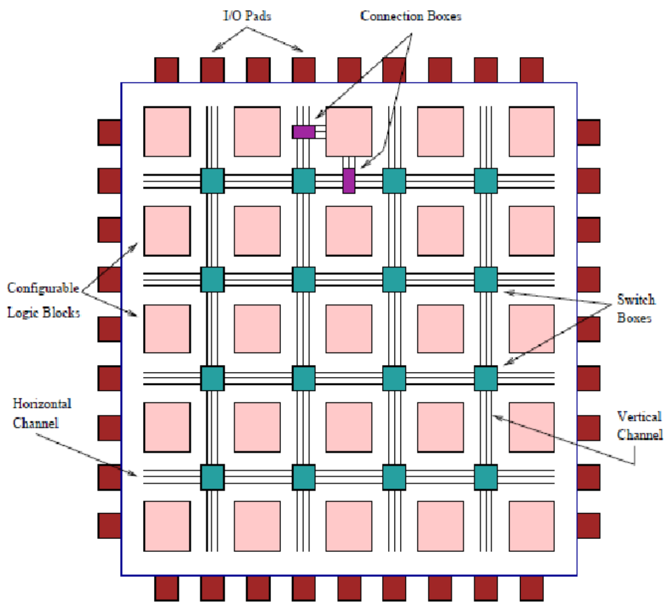


Figure 4. Overview of FPGA architecture [15]

2.2.2 Hardware Accelerators

Hardware accelerators are optimized functional blocks designed to offload specific tasks from general purpose CPUs [16]. However, the optimized and dedicated architecture on CPU can only work at a sequential instruction execution level. Since FPGAs can be highly optimized at the logic level, they are very suitable for implementing hardware accelerators.

In this thesis, all algorithms and applications were implemented on two FPGA boards, which are Stratix V A7 and Intel Arria 10 GX. Stratix V A7 FPGA board contains 2 x 4GB of 1600 MHz DDR3 memory. The Stratix V A7 was fabricated from 28 nm node which consists of 622K logic elements (LEs), 234,720 ALMs, 938,880 registers, 2,560 M20K blocks and 256 DSP blocks. The second platform Intel Arria 10 GX FPGA contains 2x 4GB of DDR3 memory. The Arria 10 GX 10AX115 FPGA, based on TSMC's 20 nm process technology, has 1,150K LEs, 427,200 ALMs, 1,708,800 registers, 2,713 M20K blocks and 1,518 DSP blocks. The layouts of Stratix V A7 and Arria 10 GX are shown in Figure 5 and 6.

Figure 5. Stratix V A7 Accelerator Board Layout [17]

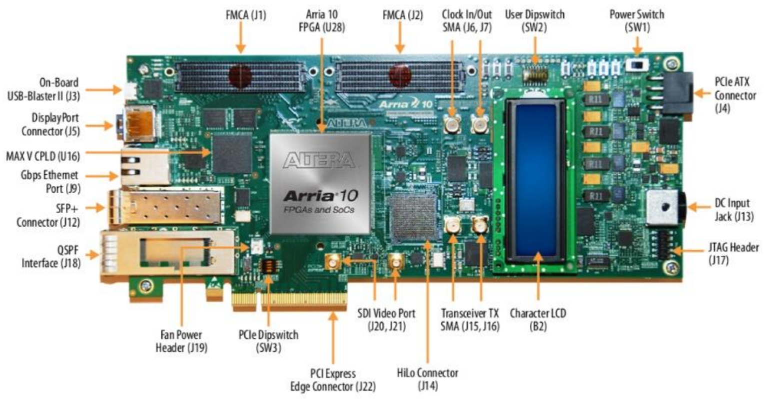


Figure 6. Arria 10 GX Accelerator Board Layout [18]

2.3 High Level Synthesis

The goal of High Level Synthesis (HLS) task is to obtain the specific functional behavior required of a system subject to a set of user constraints. HLS CAD tools provide optimized and synthesized HDL outputs at the RTL level that implement the functional behavior (subject to design constraints) given in the input specification [19]. High level synthesis is different from other types of synthesis. It is not to be confused with logic synthesis. Logic synthesis is the system specified in terms of logic equations, and must be optimized and mapped into a given technology. Logic synthesis might be implemented after high level synthesis finished in some way where logic synthesis needs to process the sorts of the decision obtained from high level synthesis.

Recently, HLS is becoming popular for real world applications. The advantages of using HLS are as follows:

- *Shorter design cycle:* Once the same type of design process is automated, designer can build the system faster. Since the design development costs too much on chip, automating more of that process will decrease the cost without doubt.

- *Fewer Errors:* Errors in design can be detected at a higher level and will not propagate to lower levels of design. This leads to reduced design debug time and cost.
- *The ability to search the design space:* Using HLS the design space and design tradeoffs in terms of area, speed and power consumption, can quickly be quickly explored. This is not feasible with traditional RTL based design methodologies.
- *The design process is self-documenting:* The decisions made in design, the reason, and the influence of these decisions can be tracked in an automated manner.
- *Availability of IC technology to more people:* HLS has the potential to enable software engineers to design hardware accelerators.

Nowadays, high level synthesis is used for accelerating FPGA design by enabling hardware developers to work at high levels of abstraction using C/C++ [20]. It allows developers to quickly explore multiple architectures through high-level directives and performs device specific timing driven schedule optimization and technology mapping for FPGAs.

2.4 Intel SDK for OpenCL

2.4.1 Overview of OpenCL

OpenCL stands for Open Computing Language, which is a C-based open stand for the programming of heterogeneous parallel devices (e.g., CPUs, GPUs, DSPs, and FPGAs). OpenCL is the first industry standard language for heterogeneous computing system [21]. OpenCL improves the speed and the responsiveness of a wide spectrum of applications in "numerous market categories including gaming and entertainment titles, scientific and medical software, professional creative tools, vision processing, and neural network training and inferencing" [22].

OpenCL specification consists of four models, which are summarized as follows:

- Platform model: The host part is the processor coordination execution and the device part are one or more processors for executing OpenCL C code. The abstract hardware model in device part called kernels allows programmers to write OpenCL C functions and execute on device.
- Execution model: Sets the OpenCL environment for host and defines how kernels are executed on the device. In this model, it includes the step of setting up an OpenCL context on the host, providing hardware mechanisms for the host, and communicating with kernels for executions on devices.
- Memory mode: Manages the abstract memory used in kernels. Same as other accelerators, the memory model also resembles current GPU memory hierarchies.
- Programming model: Developers can map the physical hardware through program in this model.

2.4.2 Intel FPGA SDK for OpenCL

IFSO is a development environment, and provides a complier and tools to build and run OpenCL applications that target Intel FPGA products [23]. In this environment, it includes "Intel's state-of art software development frameworks and complier technology with the revolutionary Intel Quartus Prime software [24]". Intel Quartus Prime software provides developers with a platform to program, implement and optimize their applications. Figure 7 shows the execution steps in OpenCL.

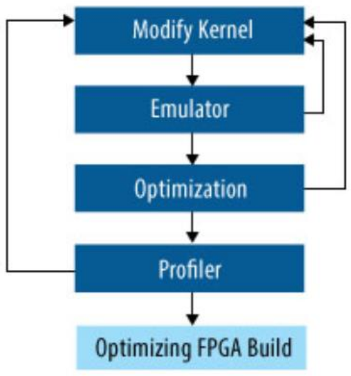


Figure 7. Execution steps in OpenCL [24]

The SDK includes IDE integration, offline complier, debugger, and other tools to develop applications. The driver/runtime package on development platforms is installed for testing. Intel FPGA SDK for OpenCL provides variety of resources and features for developers to utilize the full potential of FPGAs [25]. Parts of the OpenCL standard are implemented differently in IFSO because the target hardware is an FPGA. The main difference between IFSO and other OpenCL platforms is the kernels executed on IFSO must be compiled offline.

When processing data in parallel using IFSO, three types of parallel computation are used. First is data parallelism, where tasks of reading, convolving, pooling and writing back into memory can be done in parallel. Loop parallelism is the second type. Here the obtained processing results between loops are independent and can be executed in parallel. These operations implemented in IFSO schedules each iteration into a pipeline. The third type is task parallelism. Data parallelism and loop parallelism reduce the computing time, while the task parallelism can boost the throughput in the system. Tasks can be arranged into different kernels and executed by command queues and multiple channels separately. In task parallelism, the hardware utilization is increased. One example using task parallelism is shown in Figure 8.

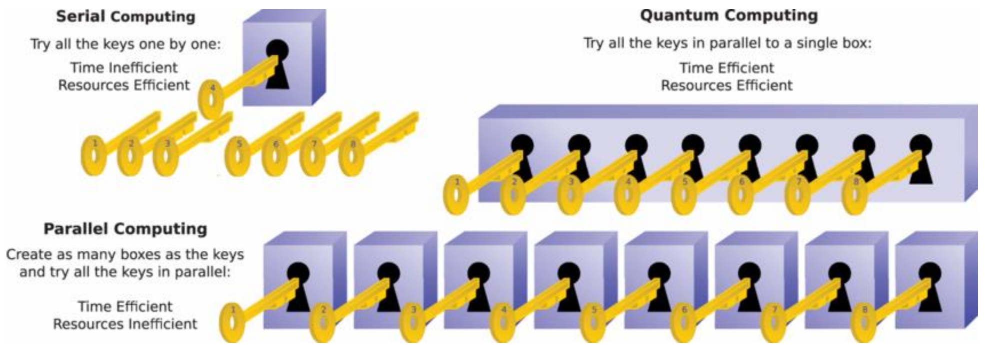


Figure 8. Example of task parallelism [26]

2.5 Machine Learning

Machine learning usually refers to the changes in systems that perform tasks associated with artificial intelligence (AI) [27]. Two main types of machine learning algorithms are supervised and unsupervised. Supervised learning algorithms work with a target variable which can be predicted by the given data. Using the given set of variables, functions or equations can be obtained for prediction. The process of getting functions is called training process, which continues until the results reach the desired accuracy on the whole training dataset. Examples of supervised learning are as follows:

- Regression
- Decision Tree
- Random Forest
- K Nearest Neighbors
- Logistic Regression

Unsupervised learning works with unlabeled data. The algorithm tries to extract useful properties from the unlabeled data. Examples of unsupervised learning are as follows:

- Clustering
- Anomaly detection
- Neural Networks

Chapter 3

Acceleration of kNN algorithm using bitonic sorting

3.1 Introduction to kNN and bitonic sort algorithms

3.1.1 Introduction to kNN algorithm

k-Nearest-Neighbor algorithm is one of the most popular machine learning algorithms [26], which was first proposed in the early 1950s. Due to high computational complexity for large datasets, implementation of kNN algorithm became feasible only in the 1960s with the availability of increased computing power. Currently kNN classifiers are widely used in pattern recognition [29]. kNN is based on analogy which makes comparison between given test datasets and training datasets that are in the same dimension. The training dataset can be separated into n attributes, and each attribute represents one n-dimensional space. kNN searches the whole dataset for k closest training samples for one unknown query point. The k closest training samples are the nearest neighbors.

To get the nearest neighbors, the distance between query points (X1) and training data (X2) must be calculated, where $X_1 = (x_{11}, x_{12}, \ldots, x_{1n})$ and $X_2 = (x_{21}, x_{22}, \ldots, x_{2n})$. The function used is Euclidean distance:

$$dist(X_1, X_2) = \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2} \tag{4.1}$$

Once all distances are obtained, they all need to be ranked. If k=1, the query point should be in the same dimension where the closest data from training dataset is. If k is more than one, the k calculated nearest neighbors must be found with smallest distance to query point.

The pseudo code for kNN is shown as follows.

Algorithm 1. Sequential kNN algorithm

```
kNN ()
Inputs:
  $X_{rc}$: training dataset array; // r is the row of the array, c is the
                              // dimension of the array
  Y: class labels of $X_{rc}$; // Y can take two values: 1 or 0
  k: number of nearest neighbors;
  x: unknown sample sequence;
Outputs:
  y: multiple clusters of size k from x;
{   // begin kNN
      for i =1 to r do
            compute distance d($X_{ic}$, x);
      end for
      sort distance d($X_{ic}$,x);
      select k reference point with smallest distance to x;
      end for
      return multiple clusters of size k from x;
}     // end kNN
```

Classifying testing samples consumes large amounts of computation time in kNN classifier. Presorting and arranging the stored datasets into search trees are two effective ways to reduce classifying execution time. The most effective method to decrease run time is parallel computation. Each of the values in training dataset can be calculated with query point independently and the distance ranking part can also be implemented in parallel as the calculated distances are independent.

3.1.2 Introduction to bitonic sorting algorithm

There are large numbers of different sorting algorithms that can be used in kNN. Most of these sorting algorithms require heavy global memory access. In heterogeneous computing with

OpenCL, the main goal is to reduce the use of global memory and find algorithm suitable for parallel computing.

Bitonic sort is not a common sorting algorithm and it is more complicated than other sorting algorithms. However, bitonic sort itself is a parallel sorting algorithm. Instead of going through the whole array repeatedly and swapping values, the bitonic sort keeps track of sorted and unsorted list separately and only swaps data when necessary. The distance values from multiple query points are processed concurrently while the sorting process itself utilizes task parallelism. In this part, additional optimizations are applied. The calculated distance values are all stored in the local memory and grouped smaller parts. There is no restore data to replace the smaller data or larger data in local memory. When all values are loaded, the data should be sorted in small parts separately and then these parts are combined repeatedly into larger groups until we form one group that represents the sorted distance values.

The pseudo code for bitonic sort algorithm is shown below:

Algorithm 2. Bitonic sort algorithm

```
Bitonic()
Inputs:
S: a random sequence;
n: number of values in S;
d: n = 2^d;
Outputs:
O: ordered S;
{     /*begin Bitonic*/
    for i = 1 to d do
      for j = i downto 0 do
            if ith bit of sequence > jth bit of sequence then
              comp_exchange_max;
                //comp_exchange_max is the sequence holding larger values
            else
              comp_exchange_min;
                //comp_exchange_min is the sequence holding smaller values
            endif
       end for
    end for
    // the sorted values are available in O now
}       // end
```

The bitonic split is a procedure that cuts one bitonic sequence into two smaller ones, where all the elements of the first sequence are less than or equal to the ones in the second. A bitonic sequence is divided between its two halves, and the nth element in each part is compared with each other. If they are out of order, they are swapped. Applying this procedure repeatedly onto the smaller lists, the result is a sorted sequence in ascending order.

Bitonic sorting network sorts n elements in $(\log_2 n)$ time. In bitonic sort, the number of training data set has to be 2^n, while not all data set would satisfy this condition. Before sorting, it is necessary to add values in the training data set to satisfy this condition. The original sequence must first be transformed into a bitonic one. Note that two numbers by themselves are a bitonic sequence, from that the sequence can be partitioned into smaller bitonic ones and then merged together.

Figure 9 illustrates the bitonic sorting process. Dark yellow colored boxes are the sorters for larger groups and light red boxes are the sorters for smaller groups. In each dark yellow colored and light red colored box, the larger values are sorted and placed at lower places (bottom of the box) and smaller values are sorted and placed at higher places. From smaller dark yellow groups to larger ones, the sequence would be divided into larger values groups and smaller values groups. Then the light red sorters make the sequence as ascending sequence (lower places) and descending sequence (higher places). For the last blue box, light red sorters rank and merge the ascending sequence and descending sequence into one ascending sequence.

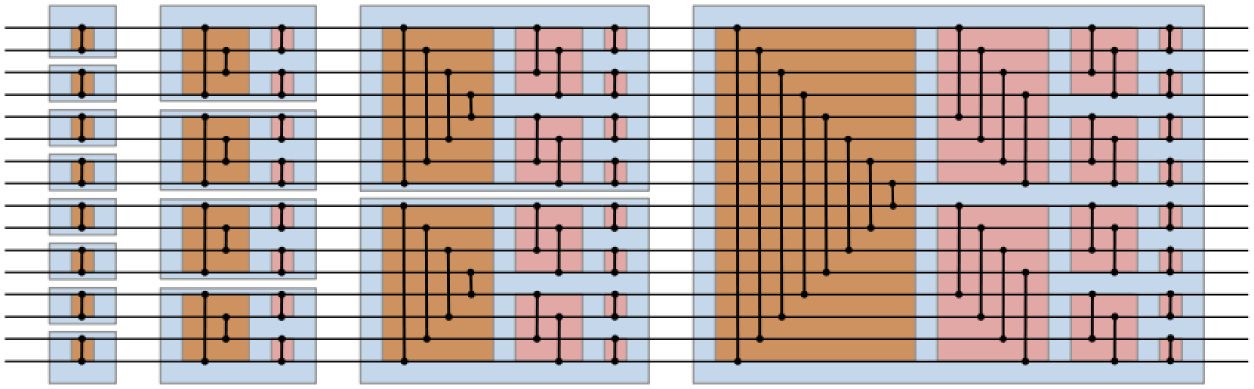


Figure 9. Bitonic sort [30]

3.2 Related Works

There have been number of research works for accelerating kNN on HCS. In addition to acceleration minimizing the hardware cost and obtaining better power efficiency are also important. Most papers focused on obtaining better acceleration performance.

H. Hussain et al. presented a dynamically reconfigurable kNN classifier implementation on Xilinx Virtex 4 FPGAs [31]. A speedup of 76X was obtained compared to a MATLAB implementation on Pentium Dual-Core E5300.

I. Stamoulias and E. Manolakos proposed implementation of kNN classifier on Xilinx Virtex 5 FPGAs [32]. It achieved 10 times slower performance than an earlier (2008) GeForce 8800GTX GPU implementation for large problem size but was considerably more power efficient and it was quoted GPU implementation claimed to be 100 times faster than CPU of its time.

Y. Pu, J. Peng and L. Huang presented an efficient kNN algorithm implemented on FPGA based HCS using OpenCL [33]. In this paper, they gave experimental results that were 148 times faster compared to an Intel Core i7-3770k CPU using bubble sort algorithm within kNN.

H. Peng and L. Huang proposed an efficient FPGA implementation for odd-even sort based kNN algorithm using OpenCL (2016) [34]. It gave 142 times speedup when compared to an Intel Core i7-3770k CPU using odd-even sort within kNN.

Qingyun Tang presented FPGA Based Acceleration of kNN using HLS which was 15 times faster when compared to a Xeon E5-2637V3 CPU [35].

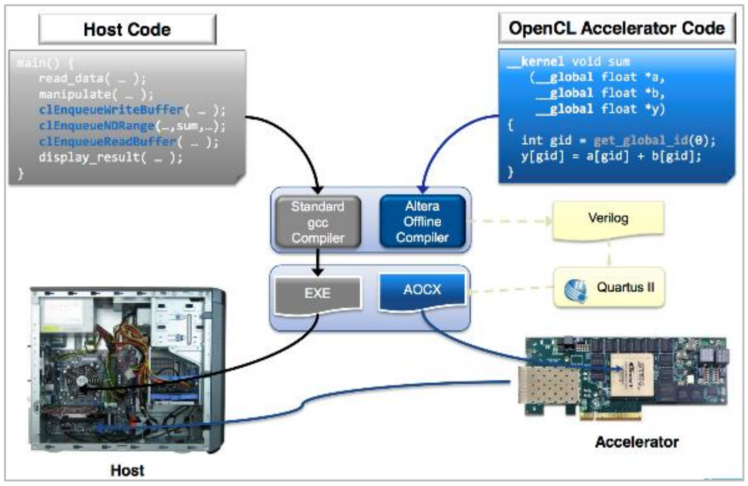
Table I illustrates the execution comparison summary from previous works.

Table I. Speed comparison summary forKNN from previous works

Reference Title	**Hardware**	**Speedup**
A dynamically reconfigurable kNN classifier implementation (2012) [31]	Xilinx Virtex 4 FPGAs	76 times faster than Matlab implementation on Pentium Dual-Core E5300
Implementation of kNN classifier (2013) [32]	Xinlinx Virtex 5 FPGAs	10 times slower than an earlier GeForce 8800GTX GPU implementation
An efficient kNN algorithm implemented on FPGA based heterogeneous computing system using OpenCL (2015) [33]	Stratix IV 4SGX530 FPGAs	148 times faster than the implementation on an Intel Core i7-3770k CPU
An efficient FPGA implementation for odd-even sort based kNN algorithm using OpenCL (2016) [34]	DE5 FPGAs	142 times faster than the implementation on an Intel Core i7-3770k CPU
FPGA Based Acceleration of Matrix Decomposition and Clustering Algorithm Using High Level Synthesis (2016) [35]	StratixV A7 FPGAs	15 times faster than the implementation on Xeon E5-2637V3 CPU and same speed with NVIDIA K620 GPU

3.3 IFSO Implementation and Synthesis of kNN

This section describes the implementation of kNN using IFSO. There are two parts of code in an OpenCL specification: Host and Device. The function of host code is to manage resources and tasks. Device part is the OpenCL computation blocks called kernels. The kernels are executed on FPGAs. Because of the parallel architecture of FPGAs, the FPGA accelerator board can provide plenty of hardware resources for the application to utilize. For many computation intensive algorithms this approach can reduce the computation time and increase the power efficiency. Figure 10 illustrates the basic usage model of IFSO.



```
main() {
   read_data( … );
   manipulate( … );
   clEnqueueWriteBuffer( … );
   clEnqueueNDRange(…,sum,…);
   clEnqueueReadBuffer( … );
   display_result( … );
}
```

```
__kernel void sum
   (__global float *a,
    __global float *b,
    __global float *y)
{
  int gid = get_global_id(0);
  y[gid] = a[gid] + b[gid];
}
```

Figure 10. Intel SDK for OpenCL use Model [36]

The kNN algorithm uses two main OpenCL kernels which implement distance calculation and sorting process.

3.3.1 Distance Calculation Kernel

The distance calculation task has several parameters. D is the dimension size of the data. T_N is the matrix of training data set. This is the most computationally intensive part of the kNN algorithm, so it is implemented in a separate kernel. For each query point, the distance calculation is independent, hence mapping the distance calculation to thread parallelism should decrease the computing time. A nested for loop is used to loop through each of the query data point and their dimensions. Each thread passes through a for-loop over the dimensions of the data, which could be unrolled to increase throughput.

To reduce the use of global memory, local memory could be used to temporarily store the calculation results in process or reference points like cache in CPU memory system. The data stored in local memory can be shared in a single work-group so the threads do not have to read them form global memory every single time when another calculation needed.

In dimensional blocked distance kernel, reference points are loaded into local memory and reused to calculate the distance by all the query data in the same work-group. All paired distance is computed using the data stored in local memory and the results are written back to global memory after the whole block is processed. The full utilization of local memory will lead reduced usage of global memory bandwidth.

3.3.2 Distance Rank Kernel

Distance ranking is a key step in kNN which utilizes sorting algorithm. There are large numbers of different sorting algorithms that can be used in kNN. Most of these algorithms require heavy global memory access. In heterogeneous computing with IFSO, the main goal is to minimize the use of global memory which can cause large latencies and maximize parallel computation on FPGA. Bitonic sort algorithm was used in the distance rank kernel. The calculated distance values are all stored in the local memory and separate the values into smaller parts, where the parts numbers are 2^n. There is no restore data to replace the smaller data or larger data in local memory. When all values are loaded, the data should be sorted in small parts separately and combine small parts to larger ones. The bitonic split is a procedure that cuts one bitonic into two smaller ones, where all the elements of the first sequence are less than or equal to the ones in the second. A bitonic sequence is divided between its two halves, and the nth element in each part is compared with each other. If they are out of order, they are swapped. Applying this procedure repeatedly onto the smaller lists, the result is a sorted sequence in ascending order. The original two sequences can be combined into one single bitonic sequence. This procedure continues until the entirety of the input has been sorted.

3.4 Result and Discussion

3.4.1 Experimental Results

In order to determine the performance obtained by IFSO for implementing the kNN algorithm, test kernels with different data dimensions were constructed to compare the 1D and 2D kernels. The 2D version gave better performance and used less FPGA reconfigurable resources, hence we use it when comparing with CPU results. Kernels optimized for data dimension sizes 32, 64, 78 and 128 were executed. The inner for loop for distance kernel with those dimensions are fully unrolled. Setting higher SIMD factor allows the complier to design hardware that could execute more work-items in parallel but will use more FPGA hardware resources. The execution performance decreases as the dimension size increases because there is not enough space in local memory to store the calculated distance values.

Tables II and III below show the performance of IFSO for kNN for three different dimensions on CPU (Xeon E5-2620), Intel Stratix V A7 and Intel Arria 10 GX. The first set of tests was conducted with varying dimension size from 64 to 128. The second table was obtained with constant dimension size of 78, but with different reference data sizes.

Table II. kNN Performance with 20480 Samples,20 Clusters and Various Dimension Sizes

Dimension Size	Xeon E5-2620 CPU execution time (ms)	Stratix V A7 execution time (ms)	Arria 10 GX execution time (ms)
64	2759.447	60.875	58.681
78	3490.068	42.836	45.364
128	12842.687	160.752	158.122

Table III. kNN Performance with 78 Dimensions, 20 Clusters and Various Data Sizes

Data Size	Xeon E5-2620 CPU execution time (ms)	Stratix V A7 execution time (ms)	Arria 10 GX execution time (ms)
1280	16.082	0.247	0.260
2560	61.113	1.018	0.984
5120	256.676	3.249	4.382
10240	898.369	11.517	10.867
20480	3494.656	43.783	45.506

Figure 11 illustrates the speedup obtained by FPGA when compared to CPU for three dimension sizes. The speedup factor increases from 64 to 78 and remains almost the same for 78 to 128. Figure 12 illustrates the speedup obtained by FPGA when compared to CPU for three data sizes. The speedup factor increases slightly as the data size increases. The local memory on FPGA board is limited and it can only store certain size of dataset. Once the dimension goes from 78 to 128, the size of calculated values is larger than the size of local memory. In this case, part of the calculated values would be stored in global memory, and they would be transferred from global memory to local memory for processing when asked. This procedure would increase the execution time.

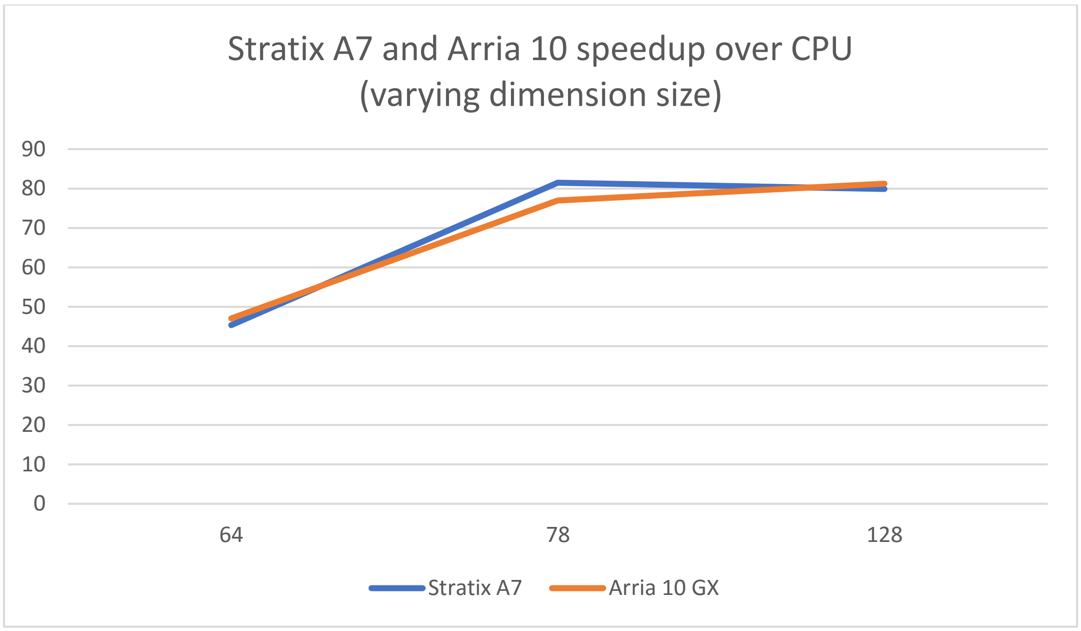


Figure 11. Speedup of Stratix A7 and Arria 10 over CPU with Varying Dimension Sizes

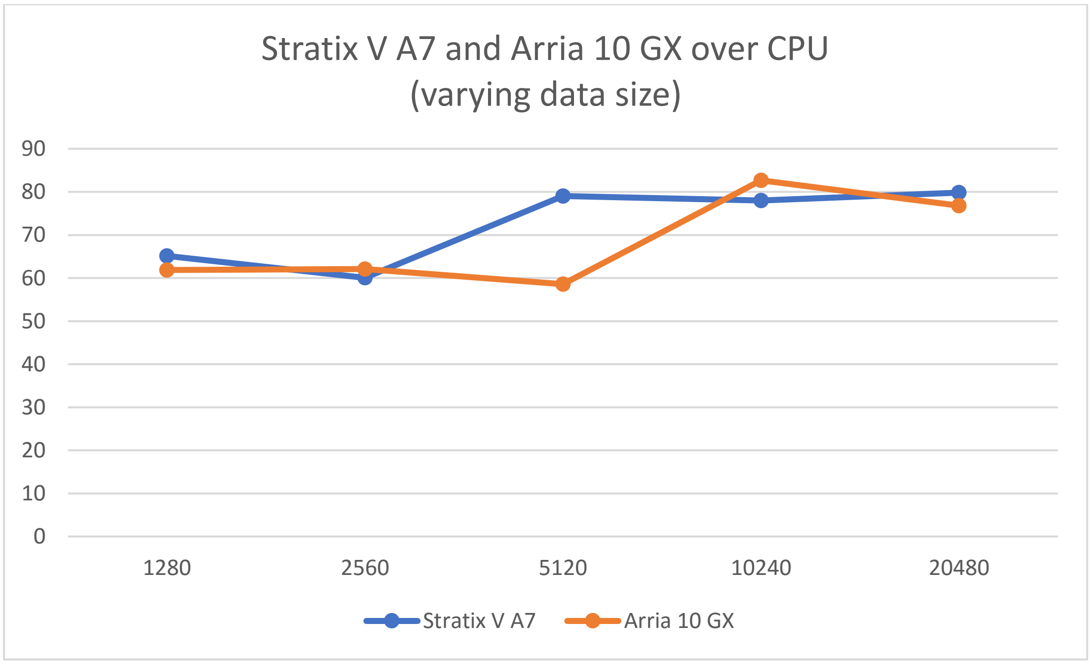


Figure 12. Speedup of Stratix V A7 and Arria 10 GX over CPU with Varying Data Size

3.4.2 Hardware Resources and Power Utilization

In our experiments, the power consumption of CPU and FPGA was measured with a power meter. The result is summarized in Table IV where accelerator power is the total power consumed by FPGA board, excluding the power consumed by the CPU.

Table IV. Power Utilization of Various KNN Implementations

System	CPU only system	CPU with Stratix V A7	CPU with Arria 10 GX
Idle Power (Watt)	106.2	126.7	127.7
Execution Power (Watt)	137.5	130.8	130.5
Accelerator Power (Watt)	-	24.6	24.3

The resource utilization of various kernels along with maximum frequencies is summarized in Table V.

Table V. Stratix V A7 and Arria 10 GX Resource Utilization and Frequency of Various IFSO KNN kernels

Kernels	Logic %	I/O pins %	DSP blocks %	Memory bits %	RAM blocks %	Kernel fmax (MHz)
64Dimension Stratix V A7	33	58	63	13	40	225.38
78Dimension Stratix V A7	29	58	100	13	35	289.26
128Dimension Stratix V A7	32	58	54	14	46	286.35
64Dimension Arria 10GX	38	16	70	18	30	251.35
78Dimension Arria 10GX	40	16	83	20	48	250.94
128Dimension Arria 10GX	38	16	79	18	59	271.58

3.4.3 Discussion

Experimental results show that the IFSO implementation of kNN algorithm with bitonic sorting gives good performance on FPGA when compared to Xeon E5-2620 CPU. For dimension sizes between 64 and 78 the speedup factor increased from 50 to 80 times. But it then levelled off when we go from 78 to 128. Similarly, the speedup factor increased with data size up to 5120 elements and then it levelled off. The limited local memory size on FPGA board may be the reason why the increase in speedup factor levels off for larger dimension and data sizes.

Chapter 4

Acceleration of kNN Algorithm Using Radix Sorting

In this chapter we present the research results obtained for accelerating the kNN algorithm using radix sorting.

4.1 Introduction to radix sorting algorithm

Radix is one of the fasted sorting algorithms [37]. Radix sort is a counting sort and fast especially for large dataset. There are two major categories for radix sort algorithms, which are strings forward from left to right, and strings backward, from right to left [38]. The well-known radix sort algorithms are radix-exchange sort [39], top-down radix sort [40], and MSD radix sort [41]. For numerical data, the main idea of radix sort is to vary each digit of the values given from input dataset. Assuming that the sorted values have $\mathfrak{i}$ digits for each value where $\mathfrak{i}$ varies from the last significant digit to the most significant digit of a number. Sort input array using count-sort algorithm according to $\mathfrak{i}$th digit. For forward scan, the input strings are split according to the first character and arranged as groups in sorted order. “Apply the algorithms recursively on each group separately, with the first character removed” [42]. After i steps of the sorting algorithm, the input strings can be sorted properly. For backward scan, the input strings are split from the last character and arrange as groups in order. Same as the forward scan, apply the sorting algorithm on all strings expect last character. The input strings would be sorted once the first character had been added into the new order strings.

The pseudo code for radix sort is shown as follows:

Algorithm 3. Radix sort algorithm **[42]**

```
Radix sort (A, d)
//It works same as counting sort for d number of passes.
//Each digit key in A[1…n] is a d-digit integer.
//(Digits are numbered 1 to d from right to left.)
       for j = 1 to d do
              //A[] -- Initial Array to Sort
              int count[10] = {0}; // for representing values from 0 to 9
              // Store count of "keys" in count[] key – it is number at digit place j
              for i = 0 to n do
                count[key of (A[i])] in pass j]++
              for k = 1 to 10 do
                count[k] = count[k] +count [k -1]
              // Build the resulting array by checking new position
              // of A[i] from count[k]
              for i =n-1 downto 0 do
                 result[ count[key of (A[i])]] = A[j]
                 count[key of (A[i])]
              // Now main array A[] contains sorted numbers according to
              // current digit place
              for i = 0 to n do
                 A[i] = result[i]
       end for(j)
end func
```

Based on the algorithm, sorting starts at the one's digit (least significant digit). Once the first loop is finished, the second loop would begin by sorting at the ten's digit. This continues until all i digits are processed.

Figure 13 shows an example illustrating the execution of radix sort algorithms.

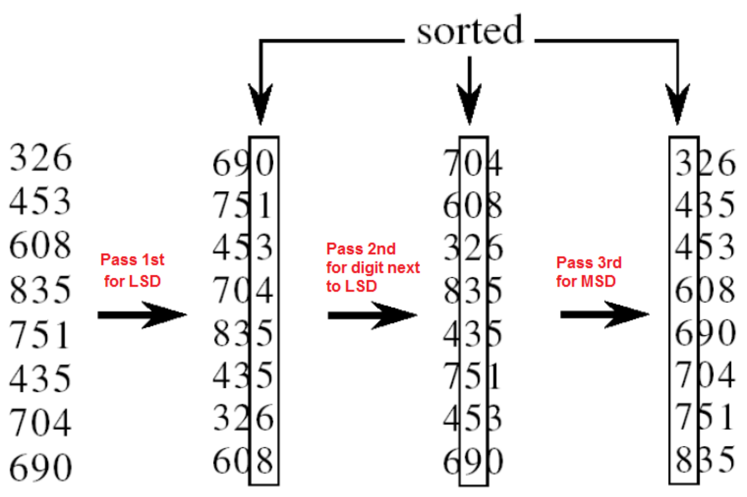


Figure 13. Example of radix sort algorithm [42]

For first pass, the least significant digit has been sorted using counting sort. Once the least significant digits from different values are the same, they should keep initial alignment. For second and third pass, the ten's and hundred's digit would be sorted in the same method.

4.2 Related Works

Previous research work on acceleration of kNN on HCS was discussed in Section 3.2. In this section, radix sorting algorithm implemented on different platforms in recent years is discussed.

Shaditalab et al [43] described the design and implementation of parallel pipelined Fast Fourier Transform (FFT), using Decimation in Frequency (DIF) algorithm on FPGAs. For N-point complex data, they have achieved an execution time of 265.45 microseconds for 1024-point FFT.

Huang et al present an empirically optimized radix sort for GPU [44]. They used on four different NVIDIA GPUs (GTX 280, 8800 GTX, 9600M GT, 9400M) and utilized empirical optimization techniques for radix sort. The maximum speedup on four platforms are 33.4%, 28%, 27.9%, and 32.6% respectively.

Khan et al presented analysis of fast parallel sorting algorithm for GPU architectures [45]. They achieved around 25X speedup on NVIDIA GTX-260 GPU comparing with 2-core Intel Q8400 CPU.

Liu and Deng presented a scalable hardware accelerator for parallel radix sort [46]. They achieved a speedup of 40X on Xilinx Virtex-5 XC5VLX110T FPGA versus DDR2-800 dual-rank 2GB CPU.

4.3 IFSO Implementation

The two main kernels in kNN algorithm are distance calculation and sorting. Here we used radix sorting in the sorting kernel.

4.3.1 Distance Calculation Kernel

This kernel is the same as that described in section 3.3.1.

4.3.2 Distance Rank Kernel

This kernel consists of two parts. The first part is utilized to rank the data obtained from local memory. The second part separates data into ten groups for storing each i step ranked distance. Once these step ranked distances are loaded, the next step is to combine these ten groups into one complete dataset. Applying this procedure repeatedly onto the next i digits, the result is a ranked sequence in ascending order.

In the second part of this kernel, the ten separated groups are used for storing current step ranked value from 1 to 10. The separated groups are stored in global memory as the number of iterations increased. As needed, these groups would be transferred from global memory to local memory for processing.

4.4 Results and Discussion

4.4.1 Experimental Results

In order to determine the performance obtained by IFSO for implementing the kNN algorithm, test kernels with different data dimensions were constructed to compare the 1D and 2D kernels. Kernels optimized for data dimension sizes 16, 32 and 64 were executed. Setting higher SIMD factor allows the complier to design hardware that could execute more work-items in parallel, but will use more FPGA hardware resources.

kNN with radix sorting algorithm was implemented using IFSO running on Stratix V A7 and Intel Arria 10 GX FPGA boards. Table VI below shows the performance for three different dimensions on Xeon E5-2620 CPU, Stratix V A7 and Arria 10 GX.

Table VI. kNN Performance with 20480 Samples, Various Dimension Sizes

Dimension Size	Xeon E5-2620 CPU execution time (ms)	Stratix V A7 execution time (ms)	Arria 10 GX execution time (ms)
16	2268.659	1586.365	1469.352
32	2869.146	1869.324	1804.657
64	3696.548	2536.544	2489.364

The FPGA platforms gave better performance in terms of execution time. Figure 14 illustrates the execution time on different platforms for the data dimensions.

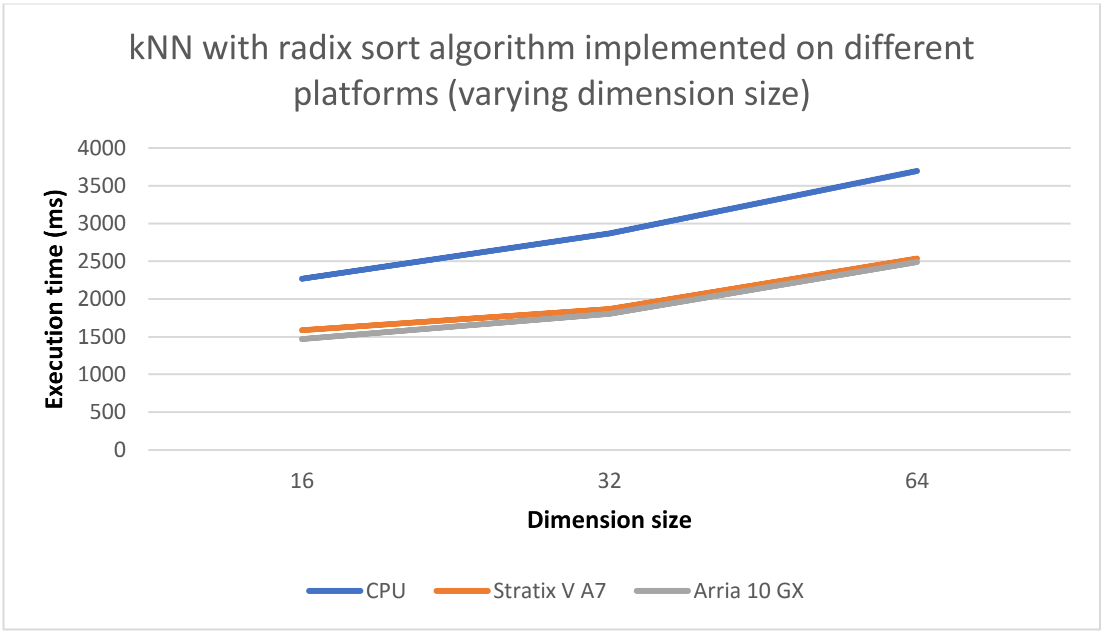


Figure 14. kNN using radix sort algorithm on different platforms with Varying Dimension Sizes

4.4.2 Hardware Resources and Power Utilization

Power meter was utilized to measure the power consumption of CPU and FPGA in performance tests. Table VII shows the power results. Accelerator Power is the total power of FPGA boards implementing on CPU exclude the power utilized on CPU.

Table VII. Power Utilization of Various KNN Implementations

System	CPU only system	CPU with Stratix V A7	CPU with Arria 10 GX
Idle Power (Watt)	106.2	127.2	127.9
Execution Power (Watt)	138.6	131.5	131.4
Accelerator Power (Watt)	-	25.3	25.2

The hardware resources utilization for implementation with maximum frequencies used are given in Table VIII.

Table. VIII Stratix V A7 and Arria 10 GX Resource Utilization and Frequency of Various IFSO KNN kernels

Kernels	Logic %	I/O pins %	DSP blocks %	Memory bits %	RAM blocks %	Kernel fmax (MHz)
16Dimension Stratix V A7	33	58	63	13	40	225.38
32Dimension Stratix V A7	29	58	100	13	35	289.26
64Dimension Stratix V A7	32	58	54	14	46	286.35
16Dimension Arria 10GX	38	16	70	18	30	251.35
32Dimension Arria 10GX	40	16	83	20	48	250.94
64Dimension Arria 10GX	38	16	79	18	59	271.58

From the tables, we can see that FPGA implementation is more power efficient than CPU. The hardware resource utilization on Stratix V and Arria 10 looks similar in percentage. However, the resources utilized on Arria 10 is more than utilized on Stratix V. Because Arria 10 is the new generation board and provides more resources when comparing to older boards.

4.4.3 Discussion

Radix sort used within kNN provides better performance on FPGA in terms of execution time and power consumption compared to a multi-core CPU. However, in distance rank kernel, radix sort algorithm uses a large number of iterations making hard to fully utilize the local memory efficiently. This limits the amount of speedup obtained.

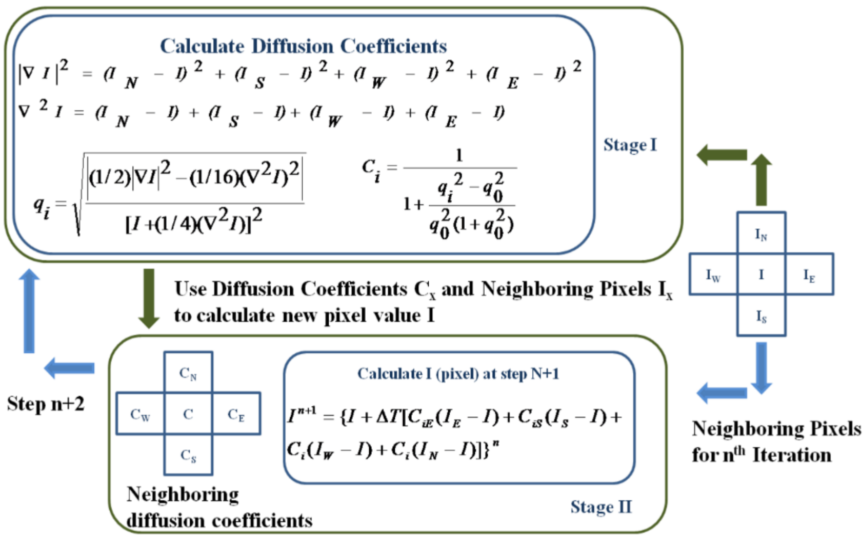
Chapter 5

Acceleration of SRAD Simulation

5.1 Introduction to Speckle Reducing Anisotropic Diffusion

Increasing chip power density has brought application specific accelerator architectures to the forefront as an energy and area efficient solution. HCS makes use of specialized hardware, such as FPGAs, to deliver higher performance and maintain the same or better power efficiency when compared to homogeneous systems [47]. Many embedded systems-on-chip (SoCs) in use today employ specialized hardware components [48]. One of the common image processing architectures of accelerators targeted at a particular ultrasound image enhancing algorithm is known as 'Speckle Reducing Anisotropic Diffusion' (SRAD). The accelerator control logic is customized to the data flow and computational requirements of the algorithm. Each computation in SRAD is complex and includes input from all neighboring pixels, and consequently, it is difficult to obtain good performance using a general-purpose processor. To achieve this goal, a high performance, low cost SRAD accelerator could be embedded into a portable ultrasound device to improve visual quality of the image.

The computational data flow of SRAD is shown in Figure 15:



Calculate Diffusion Coefficients
Stage I
Use Diffusion Coefficients C_x and Neighboring Pixels I_x to calculate new pixel value I
Calculate I (pixel) at step N+1
Step n+2
Neighboring diffusion coefficients
Stage II
Neighboring Pixels for nth Iteration

Figure 15. SRAD Dataflow [49]

Figure 15 shows the two stages and the operations performed in each stage. In the first stage, the directional derivatives for each pixel are determined, followed by the calculation of a diffusion coefficient C_i corresponding to each pixel in the input image. Stage 1 thus requires immediate neighboring pixel values to be available for each pixel currently being processed. The second stage calculates the divergence of the diffusion coefficients multiplied by the directional derivatives computed in the Stage 1 and uses the divergence to compute the new pixel value. For each new pixel value computed, Stage 2 requires immediate neighboring pixel values, and also requires immediate neighboring diffusion coefficient values computed in Stage 1 as well [49]. The computation for each active pixel I in both stages is dependent on its adjacent pixel (*IE*, *IW*, *IN*, *IS*), both in the row and column directions. Assuming row by row processing of the input image, the computations in Stage 2 depend on previous, current and future results of Stage 1. This makes any potential parallel optimization of the algorithm more complex. The algorithm is run for a number of iterations which, in most cases, is empirically determined by the user; an unnecessary number of iterations increases the computation time and produces very little additional image quality [50].

5.2 Related Research

Over the years, extensive research about speckle reducing anisotropic diffusion has been done on CPU and GPU. Y. Yu and J. Yadegar published Regularized speckle reducing anisotropic diffusion for feature characterization [51]. They put forward a new method called energy condensation integral and developed a regularized SRAD (Reg-SRAD). Reg-SRAD generates outputs using increased resolution while retaining the characteristics the SRAD-filtering speckle with regional features enhanced.

H. Kim and his research group presented Speckle reducing anisotropic diffusion based on directions of gradient [52]. In this paper, the authors added directions of gradient using Prewitt mask operations for different categories. By this method, the pixel values increased from 4 dimensions to 8 and 16 dimensions.

Evaluation of an accelerator architecture for SRAD called CPE achieved was more than 200 times speedup over CPU (Intel Core i5-2500K) and GPU (NVIDIA 8800 GTX).

5.3 Intel SDK for OpenCL Implementation

The computation of each pixel is independent of other pixels and thus could be computed using different threads, and various optimization such SIMD vectorization could be used to increase the throughput. Three phases were attempted in this research. First phase is the kernel storing pixel arrays in global memory. Second phase is to transfer the pixel array from global memory into local memory, and separating the array into small groups. In these small groups, diffusion coefficient C_i was calculated for each pixel in parallel. Third phase is to make comparison between diffusion coefficients C_i from different groups, which helps detect the speckle and recalculate the pixel value by diffusion coefficients C_i and values nearby (*IE, IW, IN, IS*). The second and third phases are similar, and the two phases share the same local memory with task parallelism. Although each phase is not memory intensive, the effect of combining the two phases is increase in memory utilization.

5.4 Result and Discussion

5.4.1 Experimental Results

When evaluating the execution speed on SRAD, frames per second (FPS) was used as an evaluation metric. It represents the frame frequency which is inversely proportional to the execution time. Once the execution time is shorter and the FPS is larger. Table IX and X below show the FPS and running time of different image sizes on CPU, Stratix V A7 and Arria 10 GX FPGAs. Table IX shows the FPS when varying image sizes from 128*128 to 512*512. The second table shows the execution time of each image size on different platforms. From Table IX and X, we note that the smaller image size processed on FPGA boards had better performance. However, more iteration computation required on larger image sizes decreased the execution efficiency when implementing on FPGA.

Table IX. SRAD Results of FPS with various Image Sizes

Image Size	CPU (FPS)	Stratix V A7 (FPS)	Arria 10 GX (FPS)
128*128	60	100	105
256*256	18	15	18
512*512	10	3	5

Table X. SRAD Execution time with various Image Sizes

Image Size	CPU execution time (ms)	Stratix V A7 execution time (ms)	Arria 10 GX execution time (ms)
128*128	1537.468	1398.249	1259.586
256*256	3569.254	3368.496	3048.364
512*512	9861.279	11359.756	10953.861

Figure 16 and 17 illustrates the speedup times when image data is processed on different platforms.

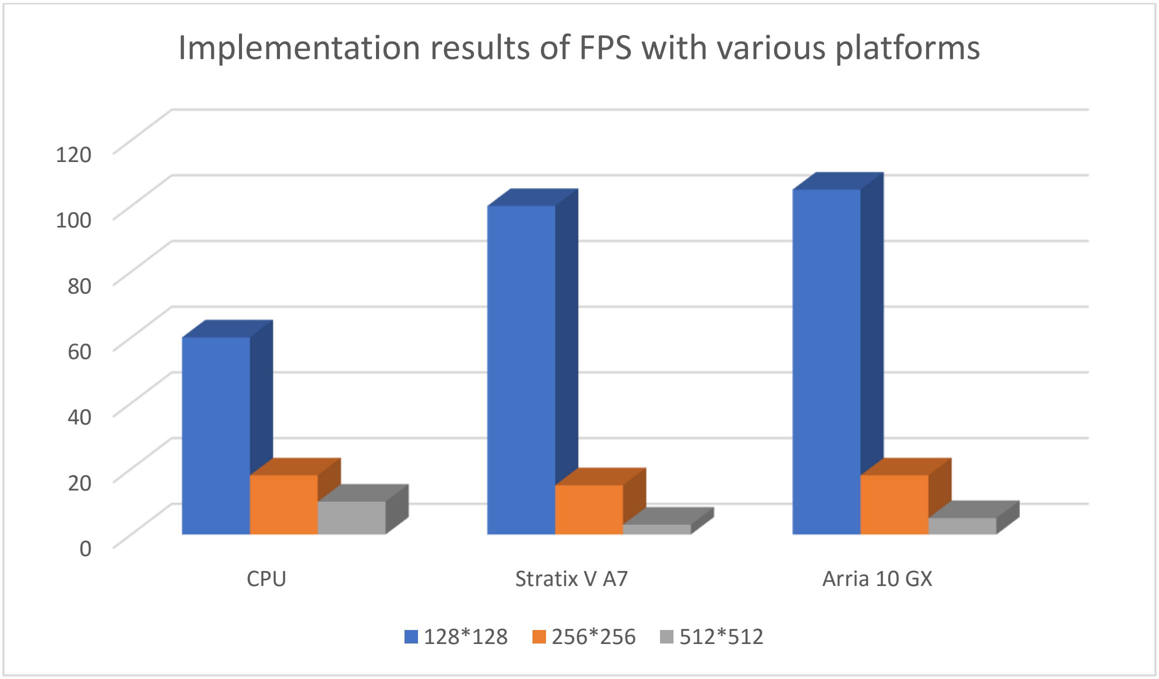


Figure 16. Results of FPS implemented on different platforms

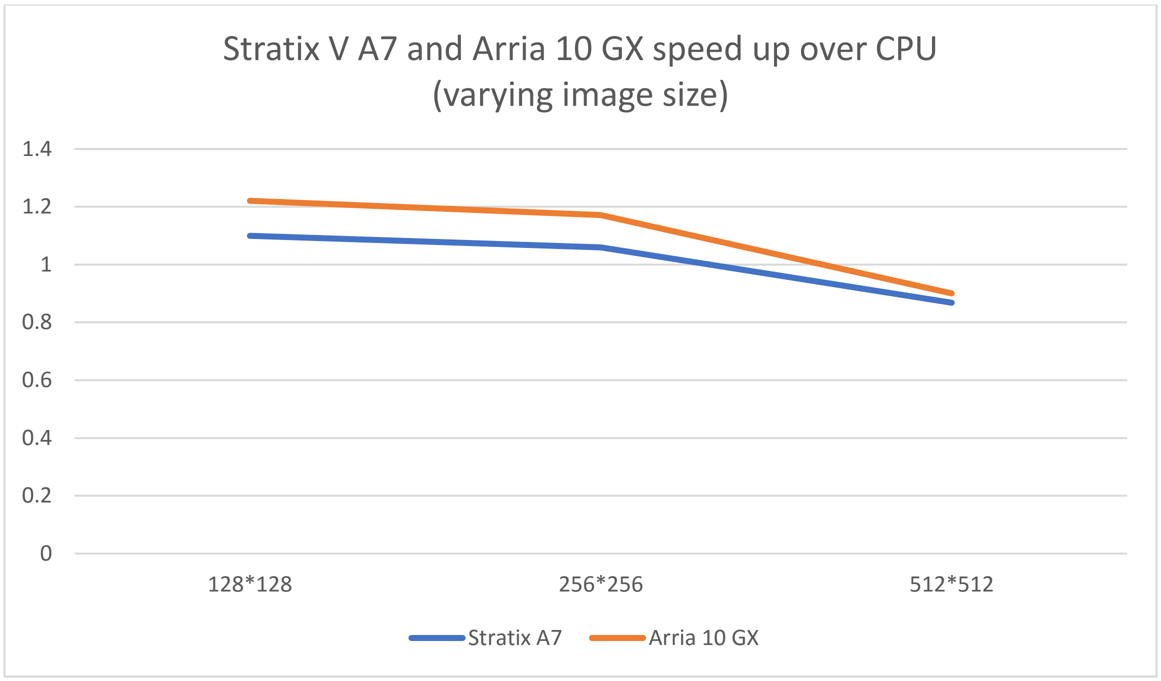


Figure 17. Speedup of Stratix V A7 and Arria 10 GX over CPU with varying image size

5.4.2 Hardware Resources and Power Utilization

In our experiments, the power consumption of CPU and FPGA was measured with power meter. The result is summarized in Table XI. In Table XI, Accelerator Power is the total power of FPGA boards excluding the power utilized on CPU. The power used on Stratix V A7 was 24.4 (130.6-106.2) and on Arria 10 GX was 25.3 (131.5-106.2). The power consumption on FPGA boards was much lower than it on CPU.

Table XI. Power Utilization of Various SRAD Implementations

System	CPU only system	CPU with Stratix V A7	CPU with Arria 10 GX
Idle Power (Watt)	106.2	126.8	127.6
Execution Power (Watt)	138.6	130.6	131.5
Accelerator Power (Watt)	-	24.4	25.3

The resource utilization of various kernels along with maximum frequencies used in the test is summarized in Table XII.

Table XII. Stratix V A7 and Arria 10 GX Resource Utilization and Frequency of Various OpenCL SRAD kernels

Kernels	Logic %	I/O pins %	DSP blocks %	Memory bits %	RAM blocks %	Kernel fmax (MHz)
128*128 pixels Stratix V A7	55	40	75	16	52	239.54
256*256 pixels Stratix V A7	60	40	83	16	42	267.52
512*512 pixels Stratix V A7	73	40	66	17	55	298.33
128*128 pixels Arria 10 GX	52	13	73	19	38	268.15
256*256 pixels Arria 10 GX	58	13	86	22	40	256.87
512*512 pixels Arria 10 GX	69	13	89	18	48	286.54

5.4.3 Discussion

During the image processing, SRAD implemented on FPGA boards shows good or similar performance compared with implementation on CPU for lower image sizes, and better power consumption in all cases. However, the intensive memory utilization on second and third phases is the vital drawback for this design. The implementation with smaller image size has fast execution time and less hardware utilization. While for higher image sizes the performance dropped fast. In this single kernel design, the diffusion coefficient C_i was calculated in second phase and reused in third phase. In these two phases, the parallel structure is not fully optimized.

Chapter 6.

Conclusions and Future Work

6.1 Conclusions

In this work, kNN using bitonic sorting algorithm, kNN using radix sorting algorithm and SRAD are implemented using IFSO targeting Intel Stratix V A7 and Arria 10 GX FPGAs. The implementation of kNN synthesized for FPGA had better performance compared to Xeon E5-2620 CPU in terms of execution time and power efficiency. Since bitonic sort is a parallel sorting algorithm it performed well during research achieving up to 80 times speedup on FPGA compared to multi-core CPU. The kNN algorithm using radix sort did not give high speedup on FPGA compared to CPU because of it's sequential nature. But the FPGA implementation still provided better power efficiency compared to a multi-core CPU. In the case of SRAD, although the FPGA power efficiency is better when compared with multi-core CPU, the execution time is just equal with CPU in smaller image size and higher in larger ones. Incomplete optimization and single kernel are the main reasons for not getting better speedup using FPGA.

Two different FPGA boards based on Stratix V A7 and Arria 10 GX FPGAs were used as hardware accelerators supported by IFSO. The process of synthesizing FPAG accelerator hardware was very quick and efficient compared to designing and interfacing FPGA hardware accelerators using traditional HDL based design methodology. It does require expertise in writing effective OpenCL programs to get the best results from IFSO in terms of execution time.

6.2 Future Work

In this research, kNN using bitonic sorting algorithm gave good speedup results on FPGAs. First, bitonic sorting algorithm is used as parallel computing structure, which is suitable for FPGAs. Second, the optimization of memory utilization increased the execution speed. kNN using radix sorting algorithm and SRAD did not give good speed performance results on FPGAs. For radix sorting algorithm, the multiple iterations are memory intensive, which cost too much time on data transmission between local memory and global memory. For SRAD, multiple kernels and better parallel structures could be implemented on FPGA to enhance performance for larger image size. Finally, the implementation on the new Arria 10 GX FPGA may perform well due to larger on chip memory size.

REFERENCES

[1] Infoworld staff, “What is Big Data? – Everything You Need to Know,” [Online]. Available: https://www.infoworld.com/article/3220044/big-data/what-is-big-data-everything-you-need-to-know.html [Accessed on: Jan-20-2018]

[2] X. Song, H. Wang, and L. Wang, “FPGA Implementation of a Support Vector Machine based Classification System and its Potential Application in Smart Grid,” 11th International Conference on Information Technology: New Generations, 2014

[3] P. Luo, K. Lv, Q. He, and Z. Shi, “A Heterogeneous Computing System for Data Mining Workflows,” Flexible and Efficient Information Handling, pp.177-189, British National Conference on Databases, 2006

[4] Benedict R. Gaster, Lee Howes, David Kaeli, Perhaad Mistry, and Dana Schaa, “Heterogeneous Computing with OpenCL”, pp.12-14, British Library Cataloguing-in-Publication Data, 2012

[5] M. Zahran, “Heterogeneous Computing: Here to say Hardware and Software Perspectives,” [Online]. Available: https://queue.acm.org/detail.cfm?id=3038873. [Accessed on: Jan-20-2018]

[6] Intel Corporation, “Intel FPGA SDK for OpenCL Overview,” [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html [Accessed on: Jan-20-2018]

[7] Wikipedia Corporation, “Machine Learning,” [Online]. Available: https://en.wikipedia.org/wiki/Machine_learning#cite_note-1 [Accessed on: Jan-29-2018]

[8] Benedict R. Gaster, Lee Howes, David Kaeli, Perhaad Mistry, and Dana Schaa, “Heterogeneous Computing with OpenCL”, pp.1, British Library Cataloguing-in-Publication Data, 2012

[9] D. Garlan and M. Shaw. “An Introduction to Software Architecture” Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Co., 1993

[10] Ahmed A. Abd-Allah, “Composing Heterogeneous Software Architectures”, Ph.D.’s thesis, Faculty of the Graduated School, University of Southern California, pp. 9-12, 1996

[11] Art and Animation studio, “What is GPU rendering?”, [Online]. Available: http://furryball.aaa-studio.eu/aboutFurryBall/whyGpu.html. [Accessed: 05-Jan-2018]

[12] U. Farooq et al, “Tree-Based Heterogeneous FPGA Architectures”, Spring Science+Business Media New York, Chapter 2, 2012

[13] Intel Corporation, “Intel® Quartus® Prime Design Software Overview,” [Online]. Available: https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html [Accessed: 01-Feb-2018]

[14] Xilinx Corporation, “ISE Design Suite,” [Online]. Available: https://www.xilinx.com/products/design-tools/ise-design-suite.html [Accessed: 01-Feb-2018]

[15] F. Piltan, O. Avatefipour, S. Soltani, and M. Ebrahimi, “Design of FPGA-Based CL-Minimum Control Unit”, International Journal of Hybrid Information Technology, Vol. 9, No. 1. pp. 101-118, 2016

[16] P. Possa, D. Schaillie, and C. Valderrama, “FPGA-based Hardware Acceleration: A CPU/Accelerator Interface Exploration”, Electronics, Circuits and Systems (ICECS), 18th IEEE International Conference, 2011

[17] P. Sutton, “Derivative Pricing on Altera’s OpenCL-enabled FPGAs”, Available: http://www.thetradingmesh.com/pg/blog/xcelerit/read/78844/derivative-pricing-on-alteras-openclenabled-fpgas. [Accessed: 05-Jan-2018]

[18] Intel Corporation, “Features of the Intel Arria 10 GX FGPA Development Kit Reference Platform”, 2015. [Online]. Available: https://www.altera.com/documentation/ewa1437420465656.html. [Accessed: 05-Jan-2018]

[19] M. McFarland, Alice. Parker, P. Camposano, “Tutorial on High-Level Synthesis”, 25 the ACM/IEEE Design Automation Coference, 1988

[20] Intel Corporation, “Intel® High-Level Synthesis (HLS) Compiler,” [Online]. Available: https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html [Accessed: 06-Jan-2018]

[21] Intel Corporation, “Altera SDK for OpenCL Programming Guide,” pp.1-3 2015. [Online]. Available: http://www.altera.com/literature/hb/opencl- sdk/aocl_programming_guide.pdf [Accessed on: Aug-20-2016]

[22] Khronos Corportaion, “The open standard for parallel programming of heterogeneous systems,” [Online]. Available: http://www.khronnos.org/opencl/, 2016, [Accessed: 06-Jan-2018]

[23] Intel Corporation, “Intel FPGA SDK for OpenCL Getting Started Guide,” [Online]. Available: https://www.altera.com/documentation/mwh1391807309901.html#mwh1391807297091 [Accessed: 06-Jan-2018]

[24] Intel Corporation, “Intel FPGA SDK for OpenCL Overview,” [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html [Accessed: 06-Jan-2018]

[25] Intel Corporation, “Altera SDK for OpenCL Programming Guide,” 2015. [Online]. Available: http://www.altera.com/lierature/hb/opencl-sdk/aocl_programming_guide.pdf [Accessed: 06-Jan-2018]

[26] P. Botsinis, S. Ng, and L. Hanzo, “Quantum Search Algorithms, Quantum Wireless, and a Low-Complexity Maximum Likelihood Itertive Quantum Multi-User Detector Design,” IEEE Access, 2013

[27] N. Nilesson, “Introduction to Machine Learning,” Department of Computer Science, [Online]. Available: http://robotics.stanford.edu/people/nilsson/mlbook.html [Accessed on: March-8 2018]

[28] X. Wu, V. Kumar, J. Quinlan, J. Ghosh, Q. Yang, H. Motoda, A. McLachlan, A. Ng, B. Liu, Z. Zhou, M. Steinbach, D. Hand, and D. Steinberg, “Top 10 algorithms in data mining,” Survey paper, Department of Computer Science, University of Vermont, USA, 1-37, 2007

[29] J. Han, M. Kamber, and J. Pei, “Data Mining Concepts and Techniques Third Edition,” British Library Cataloguing-in-Publication Data, pp. 423-425, 2012

[30] “VHDL Code for Bitonic Sorter,” [Online]. Available: https://vlsicoding.blogspot.ca/2016/01/vhdl-code-for-bitonic-sorter.html [Accessed on: Jan-20 2018]

[31] H. Hussain, K. Benkrid, H. Seker, “An adaptive implementation of a dynamically reconfigurable K-nearest neighbour classifier on FPGA,” NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2012, pp.205,212, 25-28 June 2012.

[32] I. Stamoulias, and E. Manolakos. “Parallel architectures for the kNN classifier,” ACM Trans. Embed. Comput. Syst. 13, 2, Article 22 Sept. 2013.

[33] Y. Pu, J. Peng, and L. Huang, “An efficient KNN algorithm implemented on FPGA based heterogeneous computing system using OpenCL,” in Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on. IEEE, 2015

[34] H. Peng, and L. Huang, “An Efficient FPGA Implementation for odd-even sort based KNN algorithm using OpenCL,” SoC Design Conference (ISOCC), IEEE, 2016

[35] Qingyun Tang, “FPGA Based Acceleration of Matrix Decomposition and Clustering Algorithm Using High Level Synthesis,” Master’s Thesis, Electrical and Computer Engineering Department, University of Windsor, pp.75-78, 2016

[36] Intel Corporation, “System Acceleration with FPGA using the Altera OpenCL SDK” [Online]. Available: http://www.manycoresoft.co.kr/scws15/archive/winter_school_at_snu_altera.pdf [Accessed on Jan-20-2018]

[37] T. Harada and L Howes, “Introduction to GPU Radix Sort,” Advanced Micro Devices, Inc. bonus in book “Heterogeneous Computing with OpenCL”, published 2011 by M. Kaufman

[38] A. Andersson and S. Nilsson, “A New Efficient Radix Sort,” Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium, 1994

[39] U. Manber, “Introduction to Algorithms,” Addison-Wesley, 1989. ISBN 0-201-12037-2.

[40] G. H. Gonnet and R. Baeza-Yates, “Handbook of Algorithms and Data Structures,” Addison-Wesley, 1991.

[41] J. H. Kingston, “Algorithms and data structures: design, correctness, analysis,” Addison-Wesley, 1990. ISBN 0-201-41705-7

[42] P. Mangal, “Radix sort – explanation, pseudocode and implementation,” [Online]. Available: https://www.codingeek.com/algorithms/radix-sort-explanation-pseudocode-and-implementation/ [Accessed: 14-Jan-2018]

[43] M. Shaditalab, G. Bois, and M. Sawan, “Self sorting radix_2 FFT on FPGA’s using parallel pipelined distributed arithmetic blocks” FPGAs for Custom Computing Machines, IEEE Symposium, 1998

[44] B. Huang, J. Gao and X. Li, “An empirically optimized radix sort for GPU,” IEEE International Symposium on Parallel and Distributed Processing with Application, 2009

[45] F. Khan, O. Khan, B. Montrucchio, and P. Giaccone, “Analysis of fast parallel sorting algorithm for GPU architectures’”, Frontiers of Information Technology, 2011

[46] X. Liu, and Y. Deng, “FastRadix: A Scalable Hardware Accelerator for Parallel Radix Sort,” 12th International conference on Frontiers of Information Technology, 2014

[47] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, “Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction,” In Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, pp. 81. IEEE, 2003.

[48] A. Shimpi, “NIVIDIA Introduces dual Cortex A9 based Tegra 2,” [Online]. Available: https://www.anandtech.com/show/2911 [Accessed on: Jan-05-2018]

[49] S. Nilakantan, S. Annangi, N Gulati, K Sangaiah, and M. Hempstead, “Evaluation of an Accelerator architecture for Speckle Reducing Anisotropic Diffusion,” CASES’11, October 9-14, 2011, Taipei, Taiwan

[50] Y. Yu, and S. T. Acton, “Speckle Reducing Anisotropic Diffusion,” IEEE Transactions on image processing, Vol. 11, No. 11, November 2002

[51] Y. Yu, and Joseph Yadegar, “Regularized speckle reducing anisotropic diffusion for feature characterization,” Image Processing, 2006 IEEE International Conference, 2006

[52] H. Kim, K. Park, H. Yoon, and G. Lee, “Speckle reducing anisotropic diffusion based on directions of gradient,” Advanced Language Processing and Web Information Technology, International Conference, 2008

VITA AUCTORIS

NAME:	Liyuan Liu
PLACE OF BIRTH:	Jilin, Jilin, China
YEAR OF BIRTH:	1992
EDUCATION:	South Central University for Nationalities, Wuhan, China
	Bachelor of Engineering, Automation specialty, 2010-2014
	University of Windsor, Windsor, Canada
	Master of Applied Science, Electrical and Computer Engineering 2015-2018