11-17-2019

FPGA-Based Acceleration of the Self-Organizing Map (SOM) Algorithm using High-Level Synthesis

Mohammad Abdul Moin Oninda
University of Windsor

FPGA-Based Acceleration of the Self-Organizing Map (SOM) Algorithm using High-Level Synthesis

By

Mohammad Abdul Moin Oninda

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

2019

FPGA-Based Acceleration of the Self Organizing Map (SOM) Algorithm using High-Level Synthesis

by

Mohammad Abdul Moin Oninda

APPROVED BY:

W. Abdul-Kader
Department of Mechanical, Automotive and Materials Engineering

E. Abdel-Raheem
Department of Electrical and Computer Engineering

M. Khalid, Advisor
Department of Electrical and Computer Engineering

November 15, 2019

Declaration of Originality

Abstract

One of the fastest growing and the most demanding areas of computer science is Machine Learning (ML). Self-Organizing Map (SOM), categorized as unsupervised ML, is a popular data-mining algorithm widely used in Artificial Neural Network (ANN) for mapping high dimensional data into low dimensional feature maps. SOM, being computationally intensive, requires high computational time and power when dealing with large datasets. Acceleration of many computationally intensive algorithms can be achieved using Field-Programmable Gate Arrays (FPGAs) but it requires extensive hardware knowledge and longer development time when employing traditional Hardware Description Language (HDL) based design methodology. Open Computing Language (OpenCL) is a standard framework for writing parallel computing programs that execute on heterogeneous computing systems. Intel FPGA Software Development Kit for OpenCL (IFSO) is a High-Level Synthesis (HLS) tool that provides a more efficient alternative to HDL-based design. This research presents an optimized OpenCL implementation of SOM algorithm on Stratix V and Arria 10 FPGAs using IFSO. Compared to recent SOM implementations on Central Processing Unit (CPU) and Graphics Processing Unit (GPU), our OpenCL implementation on FPGAs provides superior speed performance and power consumption results. Stratix V achieves speedup of 1.41x - 16.55x compared to AMD and Intel CPU and 2.18x compared to Nvidia GPU whereas Arria 10 achieves speedup of 1.63x - 19.15x compared to AMD and Intel CPU and 2.52x compared to Nvidia GPU. In terms of power consumption, Stratix V is 35.53x and 42.53x whereas Arria 10 is 15.82x and 15.93x more power efficient compared to CPU and GPU respectively.

Acknowledgement

Firstly, I would like to express my deepest gratitude to my supervisor Dr. Mohammed A.S. Khalid for giving me the opportunity to conduct my Masters thesis research under his supervision. I am very grateful for his patience, advice and encouragement. His encouragement helped me to the surpass difficulties that I have encountered during my research and study. I am grateful and fortunate to have him as a mentor and supervisor.

I would like to thank Dr. Esam Abdel-Raheem and Dr. Walid Abdul-Kader for taking time from their busy schedule to be part of my thesis committee and for providing insightful suggestions to improve my research.

I would like to give special thanks to Mohammad Abdul Momen for his technical assistance and suggestions that helped me to improve my research.

I am grateful to Dr. Roberto Muscedere for his continuous help in maintaining the workstations in our research lab.

I would like to thank The Canadian Microelectronics Corporation (CMC) and Intel Programmable Solutions group for supporting this research by providing us with research equipment and CAD software tools necessary for this research.

I am dedicating my research to my parents, without their continued support and care; I would not be able to finish my research on time.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ML	Machine Learning
AI	Artificial Intelligence
SOM	Self-Organizing Map
ANN	Artificial Neural Network
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
SDK	Software Development Kit
IFSO	Intel FPGA SDK for OpenCL
OpenCL	Open Computing Language
HLS	High-Level Synthesis
HLL	High-Level Language
CPU	Central Processing Unit
GPU	Graphics Processing Unit
HPC	High Performance Computing
IT	Information Technology
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
AOC	Altera Offline Compiler
CUDA	Compute Unit Device Architecture
NN	Neural Network

BMU	Best Matching Unit
API	Application Programming Interface
DPLL	Digital Phase-Locked Loop
TSP	Travelling Salesman Problem
IP	Intellectual Property
PCIe	Peripheral Component Interconnect Express
DMA	Direct Memory Access
CAD	Computer-Aided Design
DDR	Double Data Rate
SDRAM	Synchronous Direct Random Access Memory
ALM	Arithmetic Logic Module
I/O	Input/Output
DSP	Digital Signal Processing
RTL	Register Transfer Level
PLL	Phase-Lock Loop
LB	Logic Blocks
LE	Logic Elements
FLOPS	Floating Point Operations Per Second
LU	Logic Utilization
ALUTs	Adaptive Look-Up Tables
CU	Compute Unit

PE	Processing Element
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
SISD	Single Instruction Stream Single Data Stream
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data

Chapter 1. Introduction

1.1 Motivation

The demand for ML has been increasing at an exponential rate due to its ability to provide actionable insights and achieve key goals in industry and business organizations. ML algorithms have been widely used in applications involving analysis of large amounts of data in order to find patterns, make predictions, etc. One of the most important characteristics of ML is its ability to solve complex computationally intensive problems efficiently. Self-Organizing Map (SOM) [1] is an unsupervised ML algorithm and is a form of ANN. SOM, being computationally intensive, consumes a lot of hardware resources, power and takes longer execution time as the dataset size increases. The inherently parallel nature of the algorithm makes it suitable for implementation in many core and multi-core architectures. A lot of research has been done on porting ML algorithms to parallel and heterogeneous platforms [2 - 5].

High Performance Computing (HPC) platform such as GPUs and FPGAs are used as hardware accelerators to efficiently accelerate computationally intensive ML algorithms [2, 3, 6] such as SOM [7]. Currently, GPUs having the features of delivering high throughput and better memory bandwidth serves to accelerate ML algorithms. However, the large amount of power required by GPU for execution of these algorithms serves as a major drawback compared to other HPC counterparts. FPGA-based accelerators on the other hand overcome this drawback by providing high throughput with low power consumption during execution of computationally intensive algorithms.

FPGAs are programmable logic devices that provides greater flexibility and high throughput. The designs implemented in FPGA are mainly done using HDLs such as Verilog and VHSIC Hardware Description Language (VHDL) requiring extensive hardware knowledge thereby increasing the development time and cost. To exploit the potential of FPGAs fully and to make it accessible to all software developers, HLS tools such as IFSO [8] provides the opportunity to program FPGAs using HLL such as OpenCL, C and C++ without requiring extensive hardware knowledge thus efficiently accelerating computationally intensive tasks. Optimized Verilog modules are automatically synthesized into FPGA hardware binaries to be able to run on FPGA boards from HLL using the Altera Offline Compiler (AOC). Since IFSO makes the use of FPGAs accessible to all developers without requiring extensive hardware knowledge, the development and deployment of a design and the time to market for HLS-based design is significantly lower compared to RTL-based design [2, 3, and 9]. FPGAs provide the benefit of low power consumption and the fact that pipelines can be efficiently customized in FPGAs, fine-tuned for the algorithm to be accelerated, makes FPGAs a better choice compared to CPUs and GPUs.

OpenCL is the first industry standard language that supports parallel and heterogeneous computing platforms based on CPUs, GPUs, FPGAs, DSP processors, etc. and supports HLS. OpenCL was first introduces by Apple and is currently maintained and updated by Khronos Group [10] and is supported by vendors such as Intel, Nvidia, AMD, Apple, Xilinx, IBM, ARM Holdings, Qualcomm among many others. One alternative to OpenCL is Compute Unit Device Architecture (CUDA) [11], which can be implemented and deployed only in Nvidia GPUs. The IFSO consists of a host code and a kernel code. The host is responsible for initializing the device for OpenCL computation, managing hardware resources such as assigning device buffers, memory synchronization and calling the execution of kernels, whereas the computational part of the algorithm for acceleration is coded on to the kernels. The host source code is written using C/C++ and executed on the host CPU using the standard C/C++ compiler whereas the kernel code is written in OpenCL and compiled using AOC into FPGA image to be executed on the desired FPGA hardware. As the kernels take a long time to compile, they are compiled offline using AOC before the execution starts [12 - 14].

However, there are some drawbacks to using HLS. In some cases hardware synthesis using HLS may not be as efficient as hardware designed by expert skilled engineers using HDL, due to the automatic generation of the design by the software. Furthermore, due to the architectural limitations of FPGAs not all algorithms will be efficient for acceleration using HLS.

1.2 Thesis Objective

An optimized OpenCL - based FPGA implementation of SOM has been proposed in this research using IFSO with the aim to accelerate the SOM algorithm on Intel FPGAs, Stratix V and Arria 10. The major contribution of this research is the superior acceleration results obtained using IFSO compared to CPUs and GPUs. To our knowledge, this is the first research that focuses on the acceleration of SOM algorithm on FPGAs using IFSO and conducts a comparative analysis of performance and power consumption with CPUs and GPUs. The research was conducted in the following phases as shown in Fig. 1.1:

1. The fundamentals of parallel programming and IFSO were studied.
2. The SOM algorithm was implemented on CPU and GPU (using CUDA) for comparison of performance and power consumption with FPGAs.
3. Improvements were made in the baseline FPGA implementation by restructuring the operational flow of the SOM implementation, optimized for acceleration using IFSO.
4. An analysis of speedup in terms of execution time and throughput, resource utilization and power consumption was conducted between:
    a. FPGAs and CPU
    b. FPGAs and GPU
    c. Stratix V and Arria 10 FPGAs.

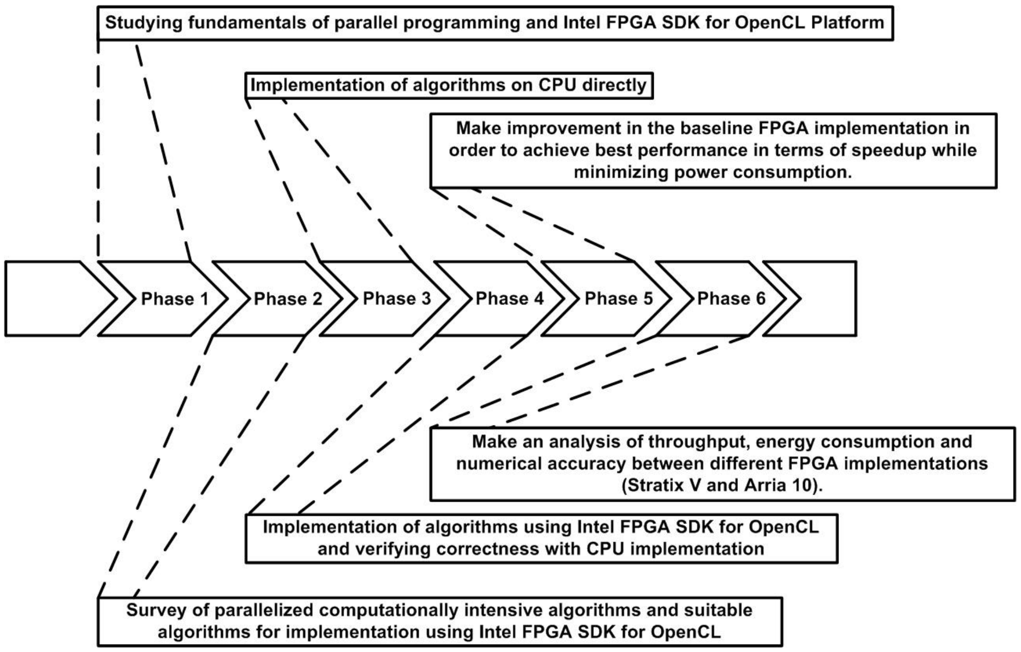

Studying fundamentals of parallel programming and Intel FPGA SDK for OpenCL Platform
Implementation of algorithms on CPU directly
Make improvement in the baseline FPGA implementation in order to achieve best performance in terms of speedup while minimizing power consumption.
Phase 1
Phase 2
Phase 3
Phase 4
Phase 5
Phase 6
Make an analysis of throughput, energy consumption and numerical accuracy between different FPGA implementations (Stratix V and Arria 10).
Implementation of algorithms using Intel FPGA SDK for OpenCL and verifying correctness with CPU implementation
Survey of parallelized computationally intensive algorithms and suitable algorithms for implementation using Intel FPGA SDK for OpenCL

Figure 1. 1 Phases of research

1.3 Thesis Outline

The remainder of this research is organized as follows:

In Chapter 2, we discuss High-Performance Computing (HPC), FPGAs, HLS, OpenCL and IFSO.

In Chapter 3, we discuss the SOM algorithm in brief and provide a review of the previously published SOM implementations targeting FPGA, CPU and GPU - based HPC platforms.

In Chapter 4, we present an optimized operational flow designed for SOM OpenCL FPGA implementation.

In Chapter 5, we present the synthesis results obtained for SOM implementation on FPGAs and compare the results with our own CPU and GPU implementations and previously published SOM research targeting CPUs and GPUs.

Lastly, in Chapter 6, we conclude with a summary of the thesis and suggestions for future work.

Chapter 2. Computing Platforms and CAD Tools

This chapter gives a brief introduction to High Performance computing (HPC) and HPC platforms, FPGAs, HLS, OpenCL and IFSO.

2.1 High-Performance Computing (HPC)

With the advancement of technology and emergence of topics like ML, Artificial Intelligence (AI), Data Mining etc. the size of data to be handled by organizations has been increasing at an exponential rate. The demand for real-time, fast an accurate processing and prediction are crucial for organizations to achieve key goals and to obtain actionable insights. The techniques and algorithms for processing the data to reach a conclusion are often computationally intensive and require large amount of computation time and power when computed on a normal CPU. HPC gives the organizations ability to compute/process data and implement computationally intensive algorithms at much higher speed compared to the traditional desktop or laptop [15, 16]. It is a subset of Technical Computing (TC) and includes Supercomputing as its member as shown in Fig. 2.1.

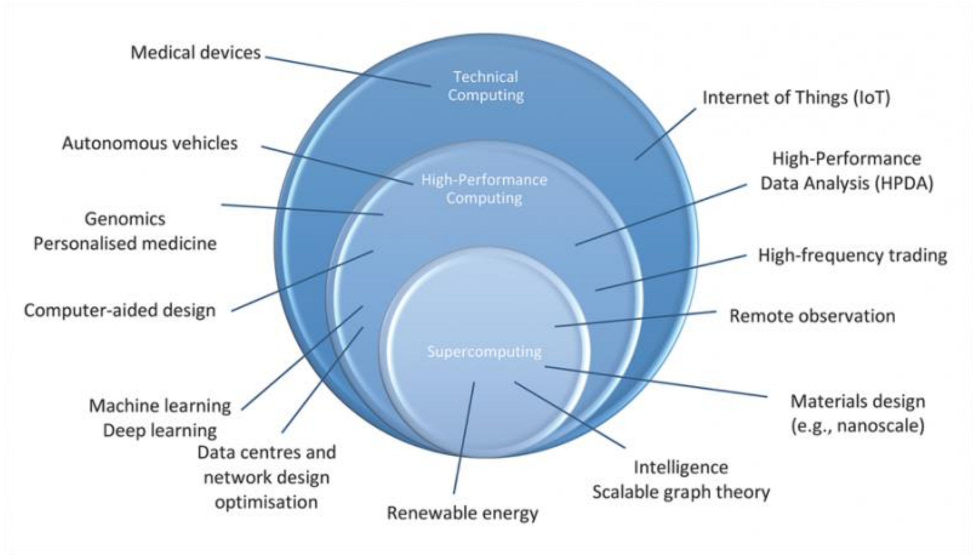


Figure 2. 1 High Performance Computing overview and applications [17]

An HPC architecture consists of compute servers called nodes networked to form clusters and a data storage facility as shown in Fig. 2.2. Each of the nodes in a cluster can communicate with each other and the data storage facility to provide maximum performance in solving computationally intensive problems. The nodes operate on the complex task in a parallel manner thereby boosting speed of execution. Hence, organizations have focused on using parallel computing and multi core designs in their Information Technology (IT) infrastructures to obtain efficient, reliable, fast and accurate solutions.

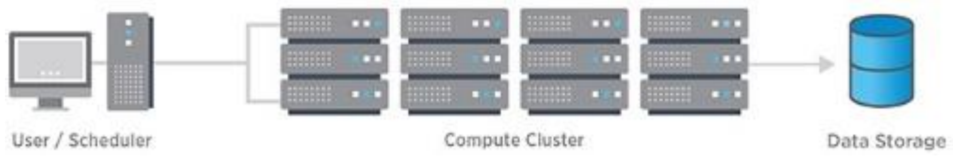


Figure 2. 2 High Performance Computing Architecture [15].

Even though parallel deployment of tasks in the HPC clusters boost performance, it may not be possible to parallelize all the algorithms/tasks. Some of the computations might run more efficiently on CPU compared to other platforms. Heterogeneous Computing (HC) overcomes this constraint by allowing the use of multiple types of processors, coprocessors and cores to execute particular tasks [18, 19]. For example, the sequential part of the task can be computed on CPU whereas the parallel part of the task can be assigned to be executed on a GPU or FPGA.

Parallel computing is the simultaneous execution of a complex task, which has been broken down into several smaller tasks and assigned to each compute unit for execution at the same time [20]. Parallel computing or parallelism can be divided into:

- Bit-Level Parallelism: Parallelism is achieved by increasing the processor word size [21]. For example adding two 16-bit numbers using 16-bit processor instead of an 8-bit processor.
- Instruction-Level Parallelism: Parallelism is achieved by instruction pipelining, superscalar execution, out-of-order execution, register renaming, speculative execution and branch prediction techniques [22]. In Instruction-Level Parallelism, several instructions are executed per clock cycle by the processor.
- Data-Level Parallelism: Parallelism is achieved by distributing the data across several compute units (i.e. nodes) for parallel execution of data. In other words is the simultaneous execution of the same task by each processor in a multi-processor environment on different distributed data [23].
- Task-Level Parallelism: Parallelism is similar in this case to data-level parallelism but in this case instead of distributing data across each processor in a multi-processor environment, this parallelism technique focuses on distributing tasks

[24]. Different tasks are assigned to each processor using the same data for execution.

Traditionally, multi-core CPUs were used as HPC clusters. CPU employs instruction-level parallelism and due to its high clock frequency, it is optimized for latency. In order to minimize memory operations, CPUs employ complicated caching schemes and have large amount of on-chip caches. GPUs on the other hand are optimized for throughput and is now being used in HPC clusters. GPUs also makes use of caches similar to CPU to minimize memory access but has far fewer caches compared to CPUs. GPU uses its high memory bandwidth and parallel execution capability to maximize throughput. The use of programming APIs such as OpenCL and CUDA have made it possible for developers to easily program GPUs for computation. However, one of the drawbacks of using GPUs is its high power consumption. FPGAs have reconfigurable resources and can be reprogrammed achieving high throughput at low power consumption, which makes FPGAs a good candidate for HPC [25].

2.3 Field-Programmable Gate Array (FPGA)

FPGA stands for Field-Programmable Gate Arrays. FPGAs provide the features of re-programmability, re-configurability and are based on Static Random Access Memory (SRAM), which is a volatile memory [26, 27]. FPGA architecture consists of I/O Banks, matrix of Configurable Logic Blocks (CLBs) and programmable switching matrix interconnecting wires as shown in Fig. 2.3.

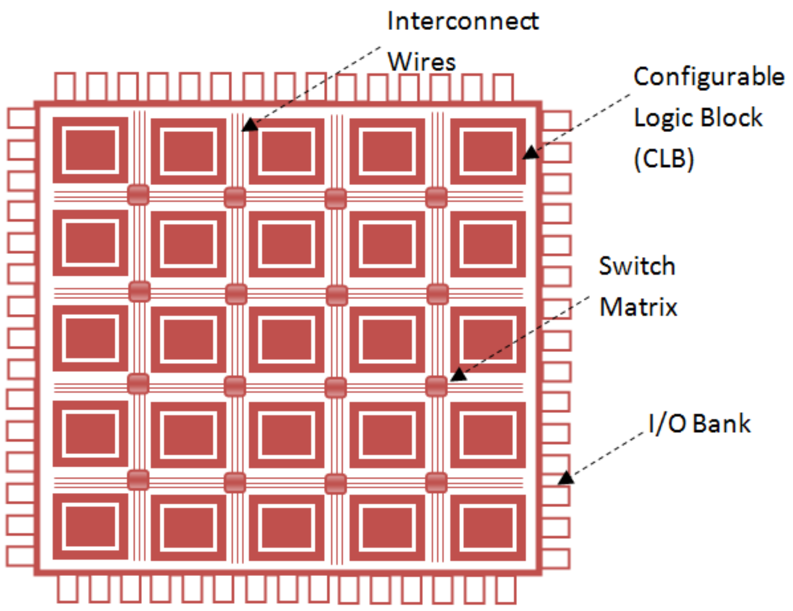
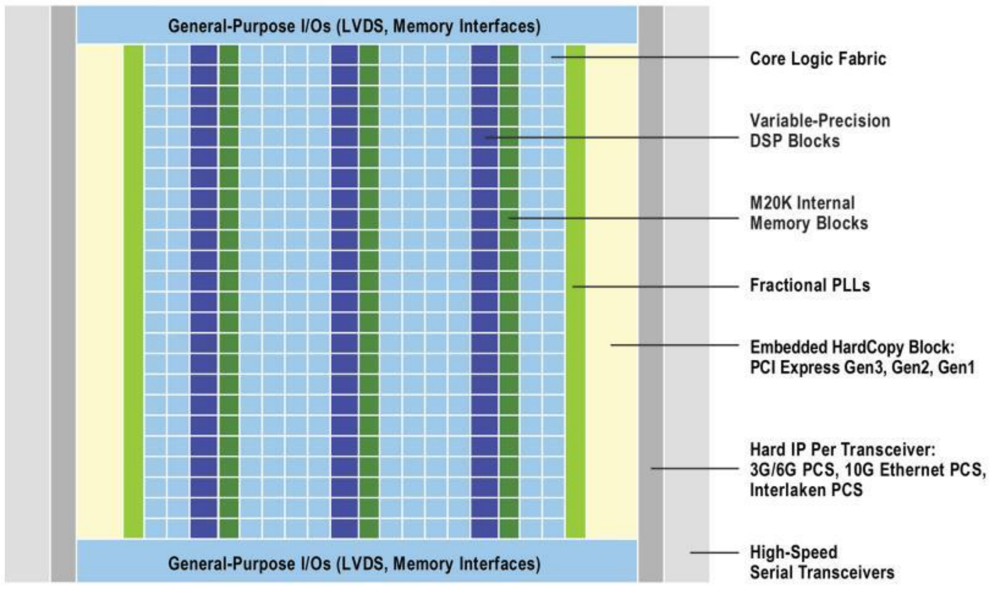
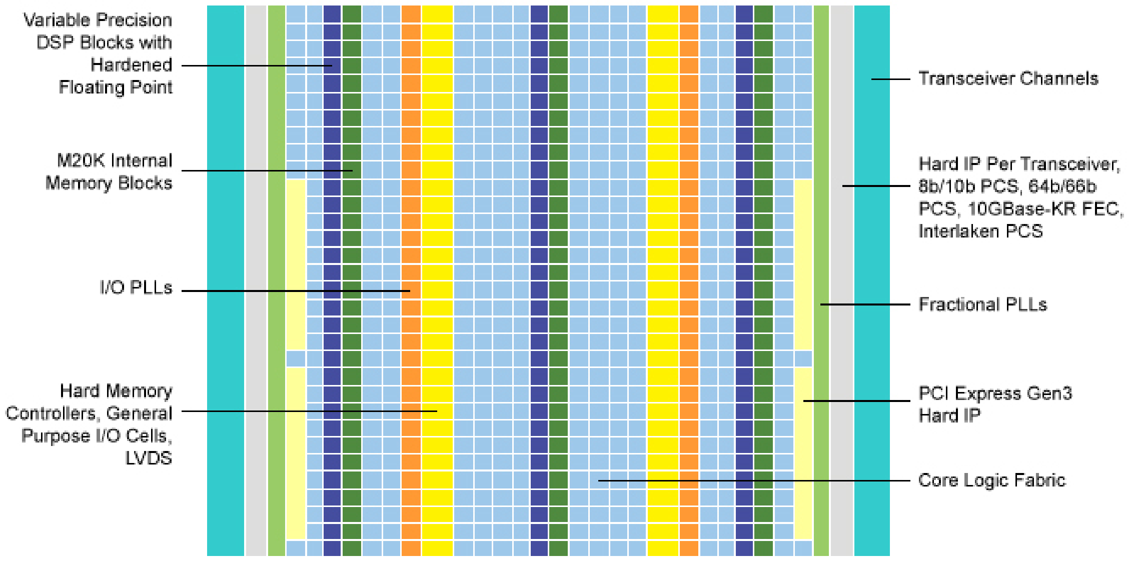


Figure 2. 3 FPGA Architecture [28]

The I/O Banks on the edge of the FPGA chip can be programmed to function as inputs, outputs, tristate buffers, differential-pair drivers, voltage logic standards, etc. Each CLB consists of a number of Logic Elements (LE), inputs and outputs. A Logic Element (LE) is composed of a Look-Up table (LUT), a Multiplexer (MUX) and a D-Flip Flop. A LUT consists of a tree of multiplexers implementing combinational logic functions, with an array of memory elements as inputs. The output of the LUT is stored in the D-Flip Flop, which can also performs sequential logic function. The MUX is used for logic selection. Since FPGA memory is volatile, the data programmed onto the FPGA memory is erased whenever the FPGA board is switched off. The basic layout of the Intel Stratix V and Arria 10 FPGA used in this research is shown in Fig. 2.4.


General-Purpose I/Os (LVDS, Memory Interfaces)
Core Logic Fabric
Variable-Precision DSP Blocks
M20K Internal Memory Blocks
Fractional PLLs
Embedded HardCopy Block: PCI Express Gen3, Gen2, Gen1
Hard IP Per Transceiver: 3G/6G PCS, 10G Ethernet PCS, Interlaken PCS
High-Speed Serial Transceivers
General-Purpose I/Os (LVDS, Memory Interfaces)

(a)


Variable Precision DSP Blocks with Hardened Floating Point
M20K Internal Memory Blocks
I/O PLLs
Hard Memory Controllers, General Purpose I/O Cells, LVDS
Transceiver Channels
Hard IP Per Transceiver, 8b/10b PCS, 64b/66b PCS, 10GBase-KR FEC, Interlaken PCS
Fractional PLLs
PCI Express Gen3 Hard IP
Core Logic Fabric

(b)

Figure 2. 4 (a) Stratix V [29] and (b) Arria 10 [30] FPGA Layout.

The FPGA boards used in this research are accelerator boards packaged in the form of a Peripheral Component Interconnect Express (PCIe) card which allows easy integration of the accelerator board into existing host system (i.e. CPU). FPGA accelerator cards are available from companies such as Terasic [31] and Bittware [32].

These vendors also provides the option for developers to design their own accelerator by changing the reference board design [33]. The accelerator cards used in these research are Nallatech 385 (Stratix V GX A7) [34] and Nallatech 385A (Arria 10 GX 10AX115) [35]. The boards include 8GB of DDR3 SDRAM memory, with x8 Gen 3 interface. The Stratix V A7 FPGA (5SGXMA7H2F35C2) is a 28nm technology consisting of 622K Logic Elements, 234,720 ALMs, 939K Registers 664 I/Os, 2560 M20K memory blocks and 256 DSP blocks. The Arria 10 FPGA (10AX115N3F40E2SG) is a 20nm technology consisting of 1150K Logic Elements, 427,200 ALMs, 1708800 Registers, 826 I/Os, 2713 M20K memory blocks and 1518 DSP blocks.

2.4 High-Level Synthesis (HLS)

FPGAs when programmed and configured properly according to the task can achieve significant increase in performance. In-order to program FPGAs, traditionally HDL such as Verilog and VHDL are used to generate hardware design at Register Transfer Level (RTL) or gate-level. However, programming the FPGA using HDL requires developers to have extensive hardware knowledge which increases the development time and cost. As HDL requires skilled developers for programming FPGAs, most of the organizations use CPUs and GPUs as they can be programmed easily instead of FPGAs.

HLS refers to an automated design process that generates the digital hardware for implementation from the interpreted algorithmic description [36]. HLS allows developers to access the full potential of the FPGA without requiring extensive hardware and debugging knowledge thus reducing development time and cost. HLS tools allows developers to use HLLs such as C, C++ or System C to synthesize their design/algorithm directly into optimized HDL for implementation in FPGAs. In [37], a detailed analysis of recent HLS tools has been provided. Table 2.1 gives an overview of some of the currently available HLS tools. For our research, we will be using IFSO.

Table 2. 1 Overview of currently available HLS CAD Tool [37]

Owner	Compiler	License	Input	Output
Intel	Intel FPGA SDK for OpenCL	Commercial	C/C++ and OpenCL	Verilog
Xilinx	VivadoHLS	Commercial	C/C++ SystemC	VHDL/Verilog SystemC
University of Toronto	LegUp	Academic	C	Verilog
Cadence	CtoS	Commercial	SystemC TLM/C++	Verilog SystemC
Mentor Graphics	DK Design Suite	Commercial	Handel-C	VHDL Verilog
Synopsys	Synphony C	Commercial	C/C++ SystemC	VHDL/Verilog SystemC
Delft University of Technology	DWARV	Academic	C Subset	VHDL

2.5 Open Computing Language (OpenCL)

OpenCL is an industry standard language, which was first, introduces by Apple Inc. and is now maintained and updated by the Khronos Group Inc. [10]. It can be executed on heterogeneous computing platforms which may be composed of CPUs, GPUs, FPGAs and Digital Signal Processors (DSPs). OpenCL is based on C99 and C++11 programming languages and defines a set of datatypes, structures and functions that augments C and C++ [38]. OpenCL provides the advantages of portability, standardized vector processing and parallel programming. It is supported and used by organizations such as Nvidia, Intel, AMD, Apple, Xilinx, Creative Technology, ARM Holdings, Imagination Technologies, Samsung, IBM, ZiiLabs, etc [39]. Computationally intensive tasks to be computed on one or more OpenCL-compliant devices are called kernels. The kernels are sent to the device(s) from the host. OpenCL uses a hierarchy of model [40] –

- Platform Model
- Memory Model
- Execution Model
- Programming Model

The platform model represents the host connected to the OpenCL devices as shown in Fig 2.5. The OpenCL devices are composed of Compute Units (CUs), which are composed of Processing Elements (PEs). Computation in OpenCL devices are done on the PEs.

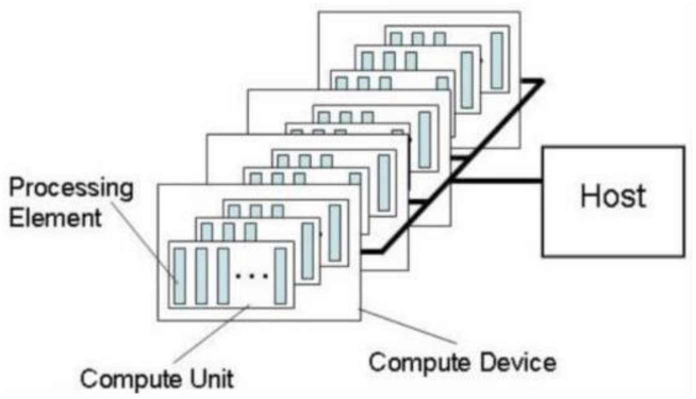


Figure 2. 5 OpenCL Platform Model [40]

Fig 2.6 shows the memory model for OpenCL. The memory model specifies four memory regions that can be accessed:

- Global Memory: The global memory can be accessed by host through PCIe and by the device. The global memory provides both read and write capabilities to all work-items in all work groups. It is the memory with largest capacity and has longer access latency and is sensitive to data access patterns.

- Local Memory: A memory region that is specific to all the work-items within a particular work-group. Local memory has lower latency compared to global memory.
- Constant Memory: It is a special type of global memory, which remains constant during kernel execution. Only the host can read/write into the constant memory, the kernels can only read data from the constant memory.
- Private Memory: It is a memory region that is assigned for a particular work-item.

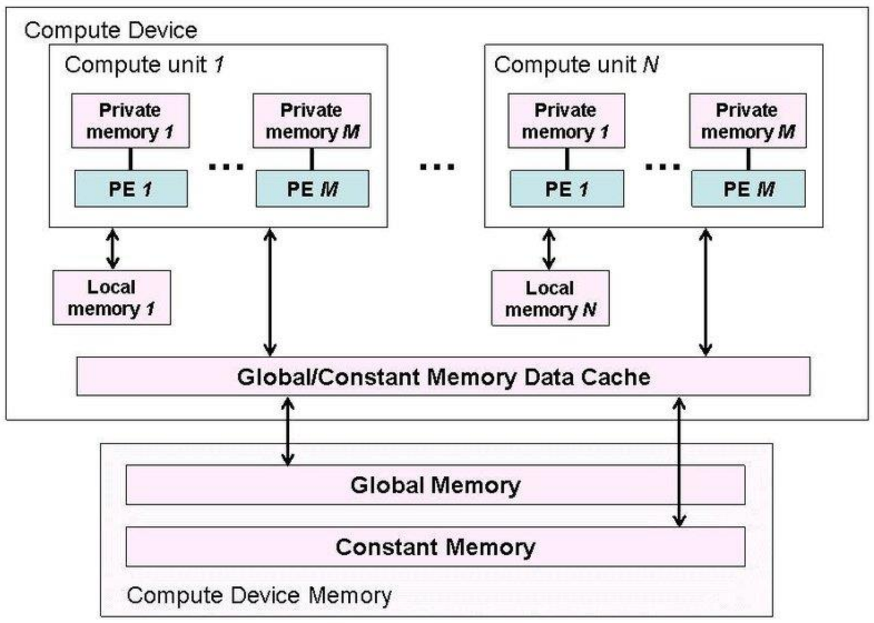


Figure 2. 6 OpenCL Memory Model [40]

The execution model consists of the kernel program and the host program. The execution is managed by the host by defining the context and the command queue. The context contains the following information: devices, kernels, memory objects and program objects. The programming model of OpenCL supports task as well as data based parallelism.

2.6 Intel FPGA SDK for OpenCL (IFSO)

IFSO is a HLS tool that enables developers to execute parallel computing programs easily and efficiently. It synthesizes the code written in OpenCL into optimized RTL Verilog code. The Verilog code can then be converted into FPGA hardware image by the Intel Quartus design software integrated with the IFSO tool. CPUs and GPUs use Single Instruction Multiple Data (SIMD) and/or Single Program Multiple Data (SPMD) model. IFSO allows FPGAs to support SIMD, Single Instruction Stream Single Data Stream (SISD), Multiple Instruction Single Data (MISD) and Multiple Instruction Multiple Data (MIMD) individually or in combination for computation. IFSO supports all features of OpenCL 1.0 and some features of OpenCL 1.2 and OpenCL 2.0 enabling the tool to accelerate algorithms efficiently.

The design flow of the IFSO is given in Fig. 2.7. In order to execute an algorithm using IFSO we need a host code and a kernel code. The host code (i.e. host.cpp/host.c) is responsible for device and host buffer initialization, transferring data from host to device for kernel execution, setting up kernel argument, calling the kernel execution command on the device and reading data back from the device to the host. The kernel code (i.e. kernel.cl) contains the computationally intensive parallel task designed for execution in the targeted FPGA board. The compilation time of kernels for OpenCL is in the order of hours. Hence, the kernel source code is first compiled using the AOC. In order to check the functionality of the kernel code, it is first compiled using an emulator integrated with the IFSO tool using the command,

aoc -march=emulator --board <board_identifier> -g device/kernel.cl -o bin/kernel.aocx -v --report

Successful emulation of the kernel indicates that the kernel.cl program has no syntax, functionality, logic and stall problems. The host code is then compiled into an executable file using the standard C/C++ compiler using the command,

make –f Makefile

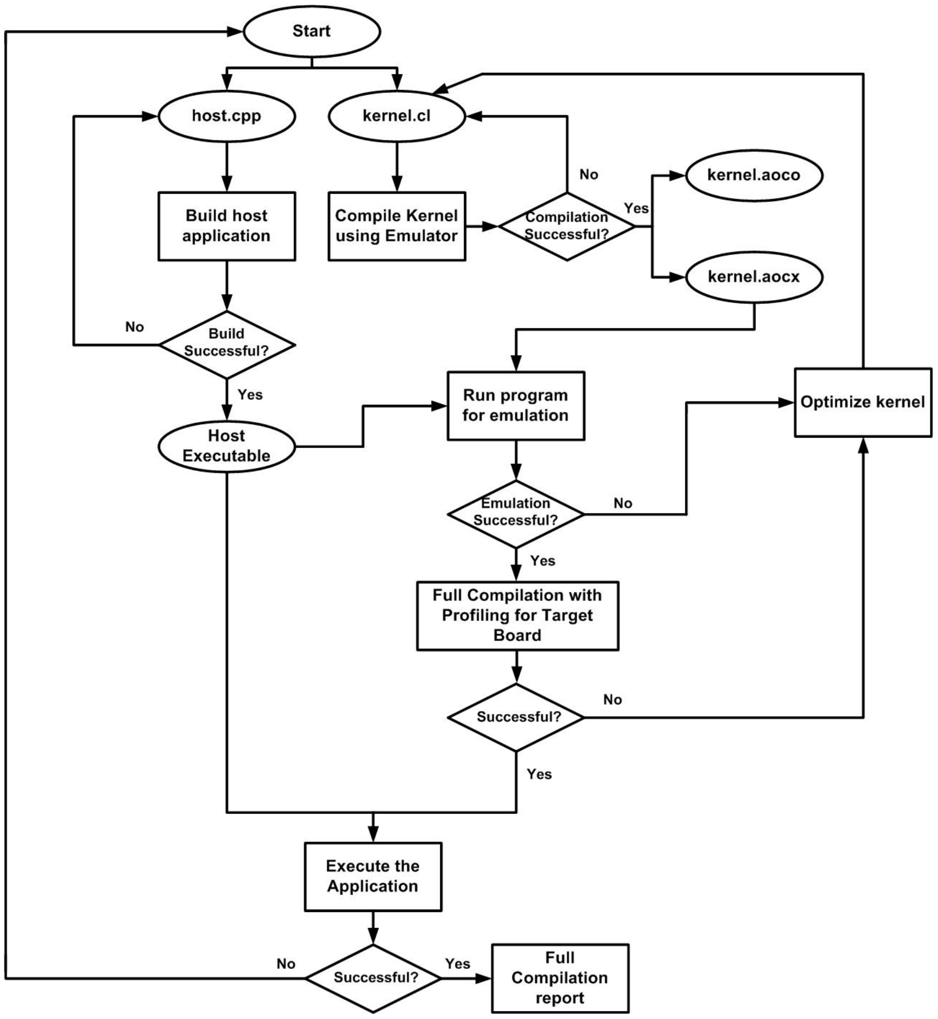


Figure 2. 7 Intel FPGA SDK for OpenCL (IFSO) Design Flow

The functionality of the program is then checked using the executable file and emulated kernel files. After successful emulation, a full compilation of the application with profiling for optimization is done in AOC using the command,

aoc --board <board_identifier> --<optimization_flags> -g device/kernel.cl -o bin/kernel.aocx -v –report

The full compilation synthesizes the kernel code into optimized RTL Verilog code and FPGA hardware image to be directly implemented onto the FPGA. The host executable and the files generated during AOC full compilation are executed by the host to run the application on the FPGA board for acceleration. The programming model for IFSO is given in Fig. 2.8. A sample template for HLS implementation using IFSO is given in Appendix A. A user perspective of the programming model has been discussed in [41].

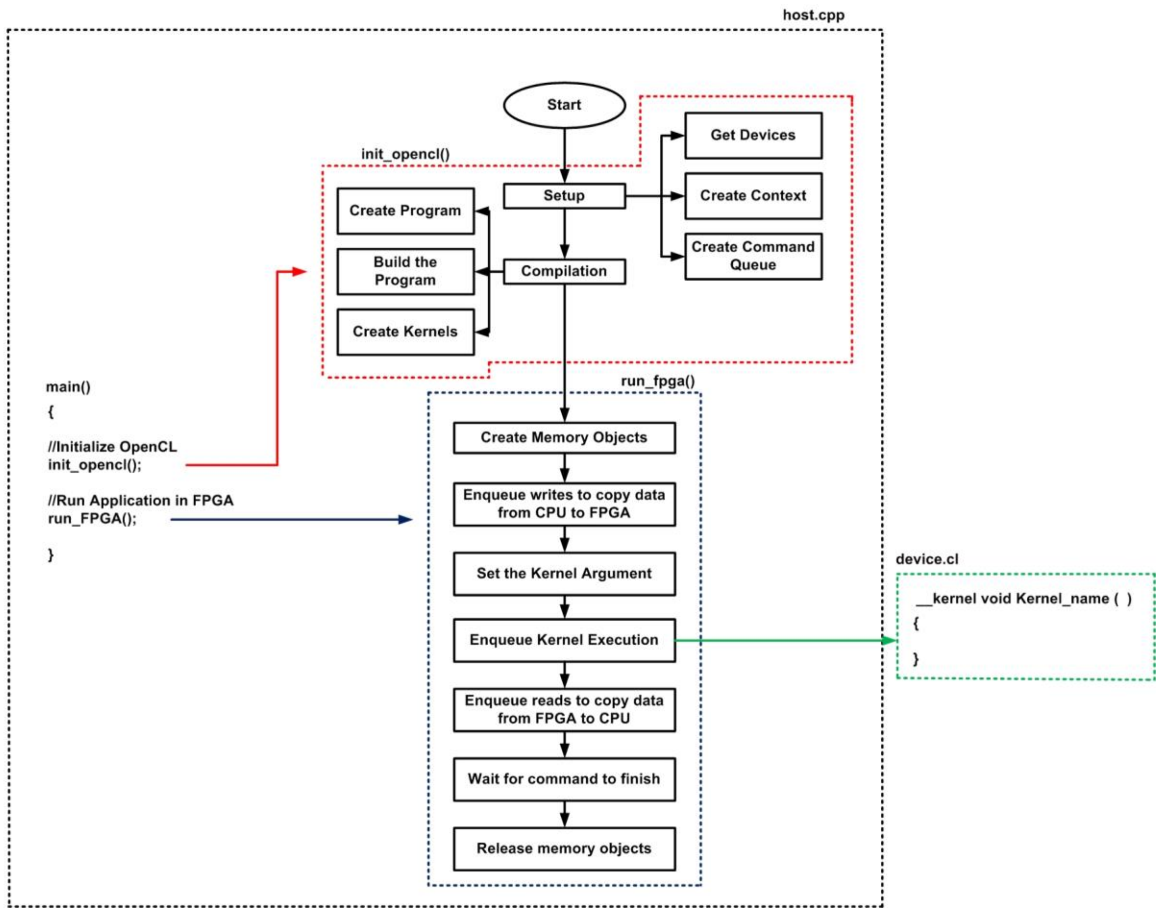


Figure 2. 8 Intel FPGA SDK for OpenCL (IFSO) Programming model.

2.6 Optimization Techniques - IFSO

The IFSO supports various optimization techniques for the acceleration/implementation of the algorithms directly onto the FPGA boards [13, 14].

- Data Parallelism: In data parallelism, work-items in a work-group are accessed by kernels using the SPMD/SIMD model. Each work-item executes the same operation on different data. In data parallelism, the highest throughput is achieved by the loops having no dependencies.
- Task Parallelism: Task Parallelism is achieved by running the kernels using command queue in a pipelined manner. Concurrent execution of the kernels by

AOC is achieved using multiple asynchronous command queues. Task parallelism requires the inclusion of explicit synchronization point. In task parallelism, the highest throughput is obtained when the application to be implemented on the FPGA is divided into multiple kernels.

- Vectorising Work-items: Vectorising allows SIMD mode execution of read/write as well as arithmetic/logic operations. It reduces memory access as the compiler creates kernel data path based on the number of vectors and increases memory read/write efficiency.
- Loop Unrolling: Unrolling loops fully or partially by including *#pragma unroll N*, where N denotes the unroll factor, before loop starts, increases the throughput of the kernel. However, increased performance comes at a cost of increased hardware resource usage as the resource usage changes based on the unrolling factor.
- Compute Units (CUs): Multiple kernel compute units creates multiple copies of the same kernel hardware for implementation simultaneously. It increases the data processing efficiency of the kernel but can cause bottlenecks in communication as the CUs share the same global memory.
- Aligning Memory: Aligning Memory allows Direct Memory Access (DMA) transfer of data to and from the FPGA increasing the data transfer efficiency. Memory alignment of host side buffers has to be at least 64-bytes aligned.
- Caching Local Memory: Local memory has high bandwidth and low latency compared to global memory. Hence, storing data from global memory to local

memory before computation starts provides the work-items easy access to the data thereby increasing throughput.

- Memory Coalescing: Memory coalescing is especially important when reading/writing data from global memory repeatedly causing performance degradation. Memory coalescing reduces the number of memory access thereby improving memory efficiency.
- Channels: Channels are First-In-First-Out (FIFO) based bus integrated in OpenCL and supported by Intel that allows efficient data transfer between the kernels in FPGA compared to the GPU, where data transfer between kernels is achieved only through global memory. FIFOs store data in on-chip memory and has high bandwidth. Channels allow the consumer kernel to launch as soon as the producer kernel has data available for transfer. However, vectorization of work-items and creation of CUs is not possible using channels.

Many other optimization techniques focusing IFSO tool is presented in "*The Best Practices Guide*" [14].

Chapter 3. Self-Organizing Map Algorithm (SOM)

3.1 Overview

In this research, we focused on the implementation and acceleration of the SOM algorithm. This chapter will first give an overview of the SOM algorithm and then will discuss some of the previous published research related to the implementation and acceleration of the SOM algorithm using HPC platforms.

3.2 Self-Organizing Map (SOM)

Self-Organizing Map (SOM) also known as Kohonen SOM or network is a form of an ANN proposed by a Finnish professor Teuvo Kohonen in the 1980s [1, 42 and 43]. It can be categorized as an unsupervised ML algorithm capable of mapping high-dimensional data into low-dimensional (i.e. usually two) feature maps and hence given the title of dimensionality reduction [44]. SOM differs from the Neural Network (NN) in the sense that unlike the NN, which consists of hidden layers, SOM does not have any hidden layer. A SOM architecture consists of two layers - an input layer and an output layer (Kohonen layer) as shown in Fig. 3.1.

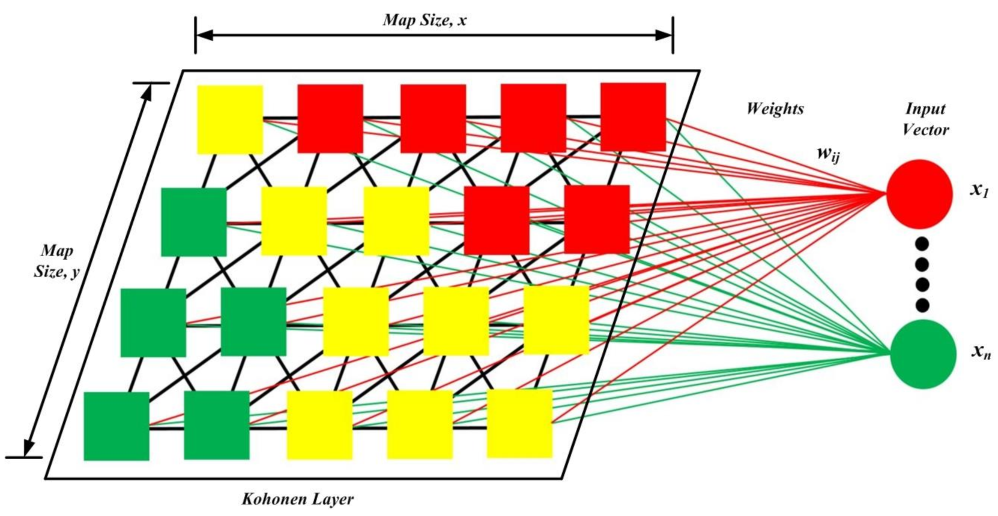


Figure 3. 1 SOM architecture.

The input layer is connected to the kohonen layer by a set of weights. The kohonen layer is a fully connected layer of neurons. The concept of SOM is neurobiological inspired and is said to have similar functionality as that of the human brain connected to the nervous system as shown in Fig 3.2. The nerve endings serve as the input layer, the nerves connected to the central nervous system and the brain represents the weight vectors and the brain represents the kohonen layer, which is responsible for mapping the signal to a particular area in the brain.

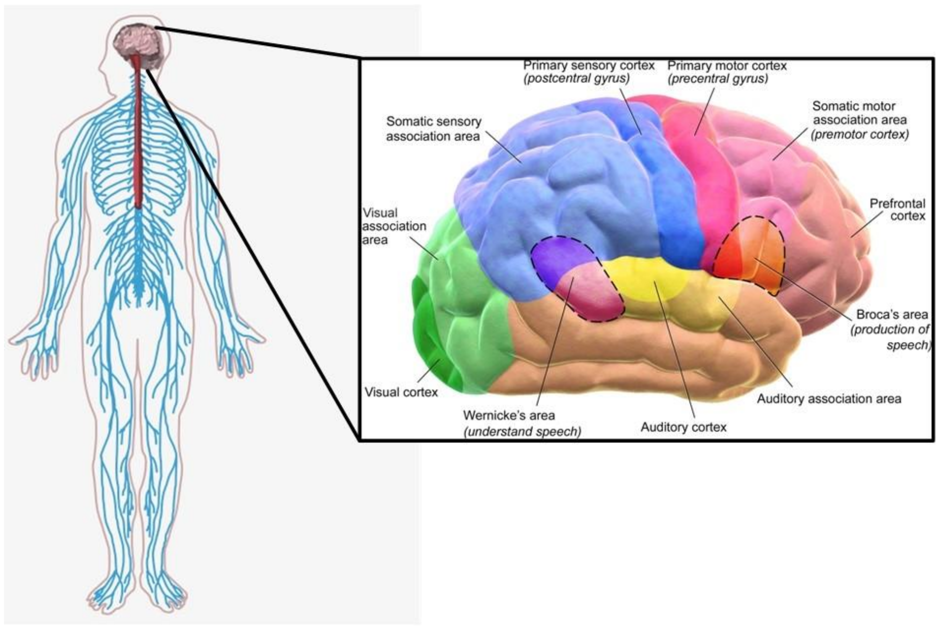


Figure 3. 2 Analogy of the SOM concept [45, 46].

An input dataset (i.e. randomly generated weight vector) is initially fed onto the SOM network. In each iteration, one sample from each input dataset, x is chosen. The distance between x and all the weight vectors of the SOM network are compared usually using the Euclidean distance. The neuron whose weight vector is closest the input vector (i.e. the computation producing the smallest Euclidean distance) is chosen as the winner neuron or the Best Matching Unit (BMU). After identifying the BMU, the weight vectors corresponding to the neurons are updated so that the BMU and its topological neighbors are moved closer to the input vector in the input space as shown in Fig. 3.3. The neurons in the output acts in a competitive manner. The neurons in the kohonen layer are said to behave in a manner such that they exhibit long-range inhibition and short-range excitation. As the iteration progresses, the neighborhood size as well as the learning rate decreases for the algorithm to reach convergence. The pseudocode of the SOM algorithm is given below.

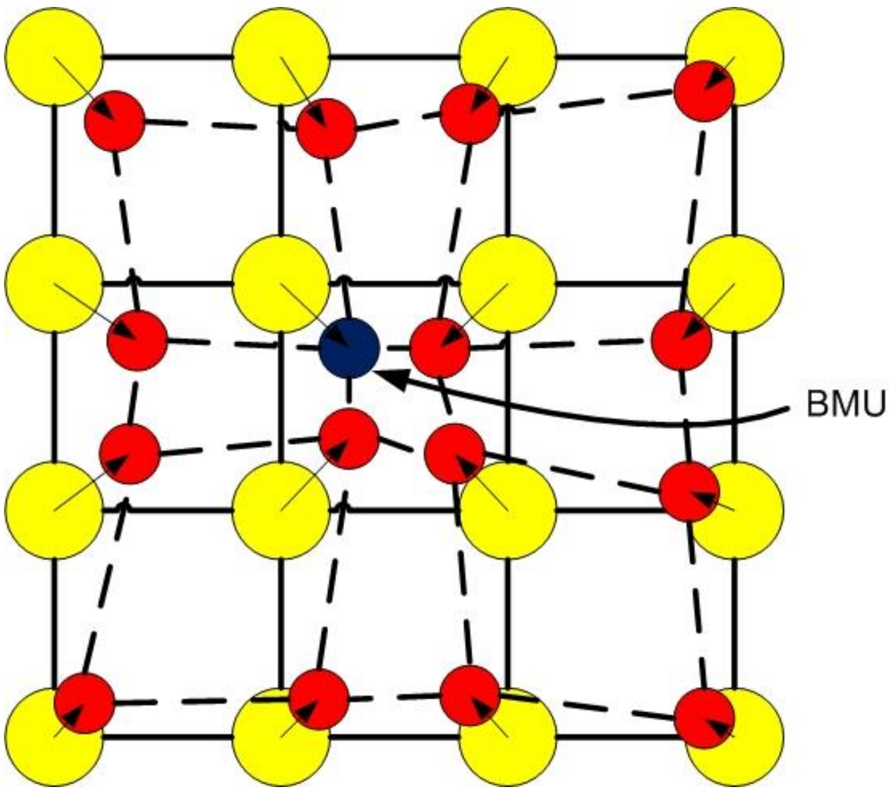


Figure 3. 3 Update stage of SOM algorithm.

Table 3. 1 Pseudo-code of SOM algorithm

```
ALGORITHM 1. Self-Organizing Map Algorithm
Input: Map Size (M), Dimension (D), Input Size, Initial Cluster,
Dataset
Output: Resultant Cluster from dataset

for all count ϵ 0 to Max_Iteration do
          //Get input vector
          for all i ϵ 0 to D
                    //Compute Euclidean distance $d_j$ between the input
                    //vector and each output node j
                    $d_j = \sqrt{\sum_{i=1}^{N} \left( x_i(t) - w_{ij}(t) \right)^2}$
                    //where,
                    //i and j are input and output nodes
                    //$w_{ij}$ represents the weight of the connected nodes
                    //t represents the time
          end for
          //Track the node that produces the smallest distance,
dist_index
          //That node becomes the Best Matching Unit (BMU) or the
          //winner neuron
          BMU ← dist_index
          for all i ϵ 0 to M*M*D do
                 //Update the weight of the nodes in the map
                   $w_{ij}(t+1) = w_{ij}(t) + \eta(t)\left(x_i(t+1) - w_{ij}(t)\right)$
                 //where,
                 //$\eta$ is the neighborhood reduction coefficient which
                 //decreases over time.
          end for
          Reduce neighborhood size
          Reduce learning rate
end for
```

The computation of Euclidean distance, finding the winning neuron/BMU and the update of weights of the neurons in the kohonen layer repeats several times until convergence is reached and thus accounts for most of the execution time. Euclidean distance step and the update step involves going through all the coordinates of the input vector and is the most computationally intensive part of the algorithm. The computational complexity of conventional SOM depends on the input vector size, N and the number of document presentation cycles (i.e. Euclidean distance computation stage), C and is given as $O(NC)$ [47].

SOM is used extensively in applications such as clustering (or classification) of satellite images [48 - 50], data visualization in finance sectors [51], modeling, probability density estimation, etc. [52, 53]. SOM being computationally intensive requires high computational time and power when dealing with large datasets.

3.3 Related Works

Extensive research has been done on the acceleration of SOM algorithm on CPU, GPU and FPGA.

In [54], a novel implementation of SOM was conducted on GPU using two different Application Programming Interfaces (APIs), OpenCL and CUDA. Two different environments were used for evaluation, a Zotac GeGorce GT 220 on AMD Athlon 64 X2 Dual Core processor 5400+ and the Sharcnet cluster Angel consisting of 11 Nvidia Tesla S1070 GPU servers (each server consisting of 4 GPUs). Speedup achieved was in the range of 3 to 32 for various map and training data size. Experimental evaluation also showed that CUDA implementation outperformed OpenCL implementation.

In [55], a massively parallel version of SOM was implemented on Intel core 2 Duo 2.66GHz platform equipped with Nvidia GeForce 9600GT achieving speedup of 44x compared to CPU. The implementation was divided into three device kernel code calls to achieve parallelism.

In [56], a SOM implementation for image pattern recognition was conducted on a Dual Core AMD Turion 64 X2 1.6Ghz platform equipped with Nvidia GeForce 6150 Go. In this implementation the images were first vectorized to form the dataset for computation, reducing the complexity and load on the GPU. The dataset generation was done on the CPU whereas the SOM computation was done in the GPU. For all the tests conducted, GPU showed significant speedup compared to CPU. The paper also highlights the overhead of data transfer between the host and the global memory which acts as a bottleneck in performance, a design consideration we will explore in our research.

In [57] a multi-pass method was used to find the location of the winner neuron and the update stage is then performed in reference to that position. Its dependence on low level textures enables efficient use of pipelines in solving large datasets. The SOM implementation was conducted on Intel Pentium 4, 2.4 GHz platform equipped with ATI 9550 and Nvidia 5700 GPU. In this implementation, GPU outperformed CPU especially for large datasets. The unique feature of this research was the accuracy resulting from the use of floating-point computation and the use of commodity graphics hardware which is easily available and widely used.

In [7], a parallel implementation of SOM was proposed using OpenCL on GPU. In this implementation, Manhattan distance was proposed compared to Euclidean distance to find the BMU. The concept of this research and the visual representation technique will be used while conducting our research. Comparison of performance of the parallel SOM implementation was conducted on AMD Operton 6366HE 1.8GHz processor, Intel core i7-2600 3.4GHz processor and Nvidia GeForce GTX 590 GPU. The output of the implementations were validated against a widely used package, SOM_PAK. OpenCL GPU implementation achieved speedup of upto 10x compared to SOM_PAK implementation on CPU.

In [58] a Digital Phase-Locked Loop (DPLL) SOM architecture has been implemented in Xilinx Virtex II FPGA using VHDL. In order to hold the value at each input vector element, the implementation uses square wave phase. The DPLL SOM design generated a small circuit and the implementation resulted in having good quantization capability. However, the proposed architecture was not as efficient in terms of speedup compared to other architectures with numerical operation.

In [59] the SOM algorithm was applied to the Travelling Salesman Problem (TSP) for a robotic mobile agent application employing embedded parallel pipeline solution (i.e. parallel pipeline architecture and parallel computation of the output and the weight update). Real-time testing of the system was achieved by generating an IP core and integrating it to a Microblaze processor bus system. The implementation was done on VHDL and the solution on FPGA showed better performance compared to CPU.

In [60], a novel SOM implementation has been proposed having the capability of identifying binary input sequence after training on Xilinx Virtex 4 FPGA (XC4VLX160) using Handel-C high-level description language. The proposed implementation utilized a novel tri-state rule during the update stage of SOM while training. The FPGA implementation achieved 30x speedup compared to conventional SOM CPU implementation and is proposed for use in fast pattern clustering and classification.

Chapter 4. Optimized OpenCL Model for FPGA Implementation

4.1 Overview

The aim of this chapter is to discuss the operation flow that we implemented in our research towards achieving high throughput and reduced power consumption using the IFSO HLS tool.

4.2 SOM Model for FPGA OpenCL Implementation

The aim of our research is to create a fully optimized FPGA implementation of the SOM algorithm for high throughput and minimum power consumption. For our research, we implemented the SOM OpenCL model using an innovative operational flow. From our comprehensive analysis, we found that if we want to address the communication bottleneck we have to change the operational flow of the SOM algorithm. Fig. 4.1 shows the operation flow of the SOM OpenCL model for FPGA implementation using IFSO. In Fig 4.1, the black arrow lines represents the instruction flow whereas the red and blue arrow lines represents the data flow from the host (i.e. CPU) to the device (i.e. FPGA) and within FPGA accelerator board respectively. The execution starts with the host side doing all the necessary computations and initializing the device buffers. The host then sends all the required data from the host memory to the device global memory through PCIe bus for SOM computation. The host then calls for the kernel execution command on the FPGA. After the execution is done, the result is then transferred from the FPGA global memory back to the host memory through the PCIe bus. After the execution is done, the result is then transferred from the FPGA global memory back to the host memory through the PCIe bus. The host, upon receiving the data from the FPGA conducts a proof of correctness test as discussed later in Section 5.7. The execution is completed if the host side implementation matches with that of the FPGA implementation.

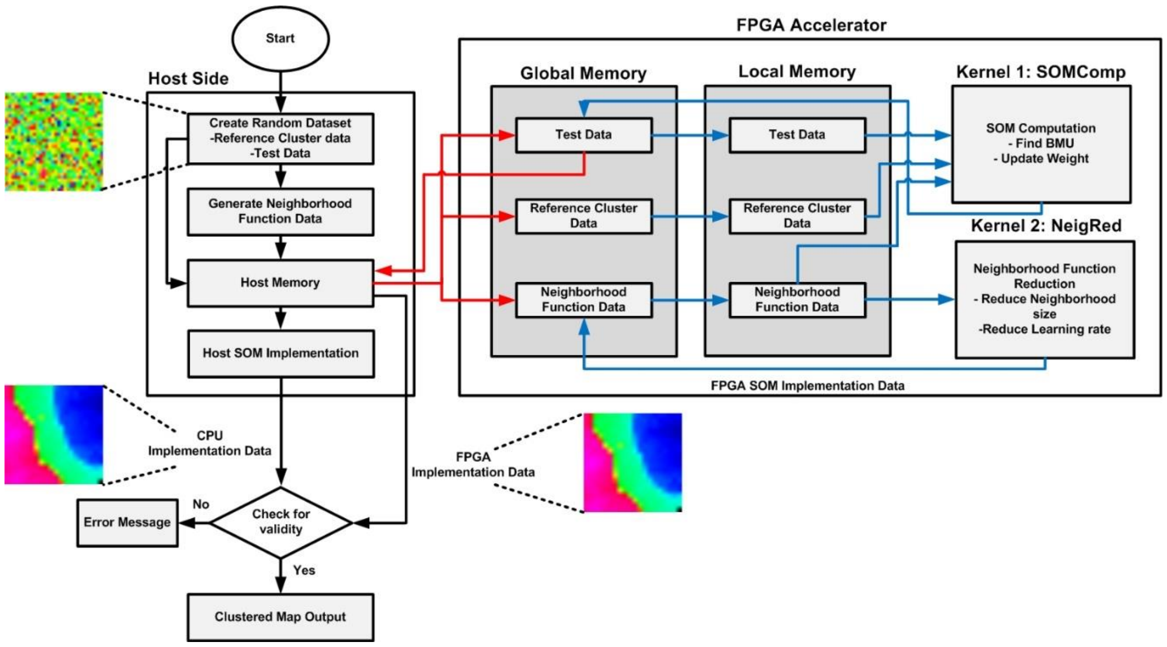


Figure 4. 1 SOM operational flow - OpenCL FPGA.

The host execution starts by first initializing the random uniform dataset in the range of 0 to 10000 having no intentional patterns in the dataset for SOM implementation. The dataset is formed into a map of predefined size (i.e. pixels) where each pixel represents a neuron in the layer having the random value acting as weights connected to each neuron. The overhead on hardware resources and memory bandwidth increases each time a kernel is added to the FPGA binary. We tried to overcome this by minimizing the number of kernels in the design thus reserving memory and hardware resources for computation. The two kernels were designed as single thread task kernels achieving task-based parallelism, due to loop and memory dependencies with no requirement of communication or data transfer between the kernels.

We decided to divide our implementation into two device kernel code calls. The first kernel (i.e. SOMComp kernel) Algorithm 2, takes care of the finding the winner neuron/BMU and the updating of weights of the BMU and surrounding neurons according to the neighborhood size and the learning rate. In contrast to the Euclidean distance equation (1) used in the conventional SOM we decided to use Manhattan Distance, equation (2) as suggested in [2 and 7] to find the BMU. Manhattan distance calculates the sum of the absolute value of the difference between two points thus requiring less resources compared to Euclidean distance computation by eliminating complex square and square root operations.

$$d_j = \sqrt{\sum_{i=1}^{N} \left(x_i(t) - w_{ij}(t) \right)^2} \tag{1}$$

$$d_j = \sum_{i=1}^{N} \left\| x_i(t) - w_{ij}(t) \right\| \tag{2}$$

$$NR = \frac{e^{\left(-\frac{(S)^2}{2*G^2} \right)}}{G * \sqrt{2\pi}} \tag{3}$$

In the update stage, the neighborhood reduction function (NR) was designed as a multiplier vector dataset, generated incorporating the neighborhood reduction size and learning rate as given in equation (3). In equation (3), S represents the neighborhood size value and G represents the gauss value, which has been predefined earlier during initialization.

Table 4. 1 Pseudo-code of kernel 1 for SOM algorithm

```
ALGORITHM 2. Kernel 1 – SOMComp
Input: Reference Cluster (R), Map Dataset (MD), Neighborhood
Reduction Dataset (NR)
Output: Resultant Cluster from Map dataset (MD)
//S = Length of one side of map, N = input size, D = Dimension, T
= Total neurons

//Load or transfer data from global memory to local memory
pragma unroll S
for all i ϵ 0 to T*D do
       local_MD ← MD
end for
pragma unroll S
for all i ϵ 0 to N*D do
       local_R ← R
end for
pragma unroll S
for all i ϵ 0 to S do
       local_NR ← NR
end for
// Start SOM Computation
for all count ϵ 0 to N*D; count = count + D
       //Manhattan distance
       pragma unroll D
       for all i ϵ 0 to D do
              dist += |local_MD – local_R|
       end for
       temp_winner_dist ← dist
       for all i ϵ 0 to T*D; i ← i +D do
              for all j ϵ 0 to D + i do
                     dist += |local_MD – local_R|
              end for
              temp_dist_vect ← dist
       end for
       //Finding the BMU
       pragma unroll S
       for all i ϵ 0 to T*D do
              if(temp_dist_vect < temp_winner_dist) do
                     BMU ← i
                     winner_distance ← temp_dist_vect
              end if
       end for
```

```
        //Update weights of BMU and neighboring neurons
        for all i ϵ 0 to T*D do
                local_MD ← local_MD - ((local_MD - local_R) * local_NR)
        end for
end for
//Load or transfer data from local memory to global memory
for all i ϵ 0 to T*D do
        MD ← local_MD
end for
```

The second kernel (i.e. NeigRed kernel) Algorithm 3, takes care of reducing the neighborhood size and the learning rate for next iterations of the SOM implementation by simply shifting all the element in the NR vector to the element before it and filling the end of the vector with zero.

Table 4. 2 Pseudo-code of kernel 2 for SOM algorithm

```
Algorithm 3. Kernel 2 - NeigRed
Input: Neighborhood Reduction Dataset (NR)
Output: Neighborhood Reduction Dataset (NR)
//S = Length of one side of map

//Load or transfer data from global memory to local memory
pragma unroll S
for all i ϵ 0 to S
        local_NR ← NR
end for
Pragma unroll S
for all i ϵ 1 to S
        temp ← local_NR[i]
        local_NR[i - 1] ← temp
end for
local_NR[S-1] ← 0
```

```
//Load or transfer data from local memory to global memory
pragma unroll S
for all i ϵ 0 to S
       NR ← local_NR
end for
```

Various other optimization techniques were also implemented in order to achieve better throughput at the cost of increased resource utilization as suggested in [13][14]. Parallelism was achieved by pipelining and loop unrolling techniques (i.e. allowing the kernel more operations per clock cycle). Floating-point operations were optimized using the balanced-tree floating-point implementation and rounding operations to reduce the amount of resources consumed achieving a fused floating-point operation. Buffer transfer efficiency was ensured by initializing the host buffers to be at least 64-bytes aligned and using Direct Memory Access (DMA) to transfer data to and from FPGA. Unnecessary memory dependencies between non-conflicting load and store units were prevented by instructing the AOC to avoid pointer aliasing. Memory bandwidth was maximized by manually partitioning global memory buffers to optimally control memory access by the device. Bottleneck in performance was further mitigated by transferring data from the global memory to the local memory before computation inside kernels. Global memory exhibits long access latency whereas local memory has much lower access latency and far higher bandwidth compared to global memory thus aiding in improved performance of the SOM algorithm.

Chapter 5. Experimental Results and Analysis

5.1 Overview

The aim of this chapter is to evaluate the acceleration results obtained for the proposed SOM algorithm on IFSO. We start by outlining the experimental setup for the implementation, dataset used, synthesis results and finally we compare the performance of our SOM-FPGA implementation in terms of speedup (execution time and throughput) and power consumption against other SOM implementations using CPU and GPU presented in previously published research.

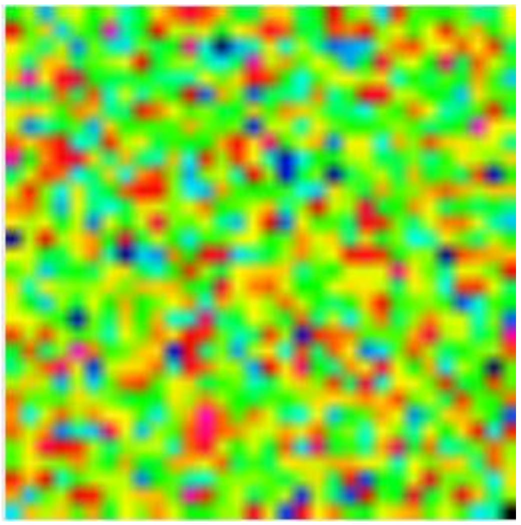
5.2 Experimental Setup

For our research, we used IFSO [61] as HLS, Computer-Aided Design (CAD) tool. The two FPGA development boards used in this research are Nallatech 385 (Stratix V GX A7) [34] and Nallatech 385A (Arria 10 GX 10AX115) [35]. The boards include 8GB of DDR3 SDRAM memory, with x8 Gen 3 PCIe interface. The Stratix V A7 FPGA (5SGXMA7H2F35C2) is based on 28nm technology, consisting of 622K Logic Elements (LEs), 234,720 ALMs, 939K Registers, 664 I/Os, 2560 M20K memory blocks and 256 DSP blocks. The Arria 10 FPGA (10AX115N3F40E2SG) is based on 20nm technology consisting of 1150K LEs, 427,200 ALMs, 1708800 Registers, 826 I/Os, 2713 M20K memory blocks and 1518 DSP blocks. To compare FPGA performance against CPU performance we are using results from [7] using AMD Operton 6366HE processor (64 cores, 32nm process) @1.8GHz with AMD's Turbo charge technology and Intel Core i7-2600 (4 cores, 32nm process) @3.4GHz and AMD Athlon II 170u processor (45nm process) @ 2.0GHz [62]. To compare FPGA performance against GPU performance we are using Nvidia Quadro K620 (28nm process) [63] having 2GB of DDR3 memory, 29.0

GB/s of memory bandwidth and 384 Nvidia CUDA® cores and implementation result from [7] using Nvidia GeForce GTX 590 (40nm process) GPU.

5.3 Dataset Generation

For our research, we conducted multiple tests with different map sizes (8x8 - 64 neurons, 12x12 – 144 neurons, 16x16 – 256 neurons, 20x20 – 400 neurons and 24x24 – 576 neurons respectively), input sizes (1024, 2048, 3072, 4096 and 5120 respectively) and dimensions (3, 4, 5 and 6 respectively). We generated random floating-point data from 0 to 10000(unsigned) and used it as dataset for computations. We selected different map sizes, input sizes and dimensions in order to evaluate its effect on FPGA performance compared to CPU and GPU. The dataset was generated in order to have similar parameter setting as that proposed in previous research, which will allow us to compare our FPGA implementation with the results published in [7]. Fig 5.1 shows an example of the dataset visualized by pixels using technique mentioned in [7 and 64] where each pixel in the map represents a neuron.


Sample 1

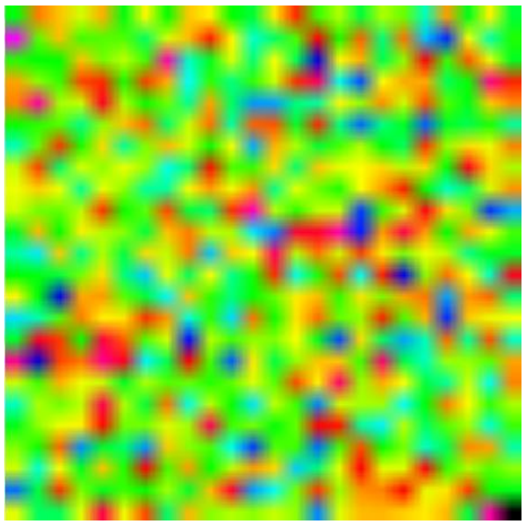

Sample 2

Figure 5. 1 Visual representation of sample dataset for computation.

5.4 Synthesis Results

The compilation of the kernels for execution on FPGA is done using the AOC, which takes on average around 4 hours. Fig. 5.2 shows the top-level block diagram of the inferred RTL circuit obtained using the netlist viewer option in the Intel Quartus Prime Pro software. The box highlighted in blue indicates the connection to the kernel system. The box highlighted in red is a magnified view of the kernel system containing the kernel blocks (i.e. Kernel 1 and 2). The logic surrounding the SOM kernels are used to communicate with DDR memory and the host. It is very difficult to visualize the detailed kernel hardware due to the complexity of the IFSO generated RTL structures. The netlist viewer feature of the software was also utilized to view the State-Machine view of our SOM FPGA design as shown in Fig. 5.3.

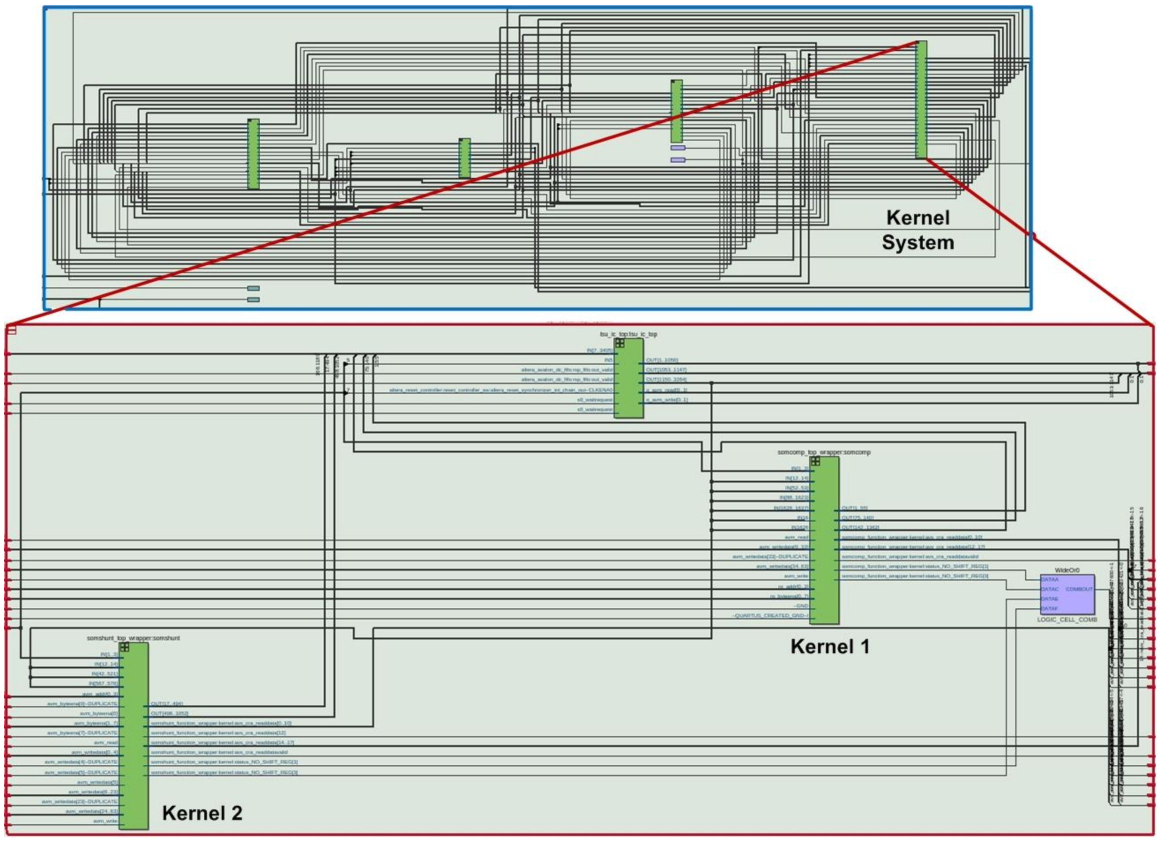


Figure 5. 2 Block Diagram of Inferred RTL Circuit for SOM FPGA implementation.

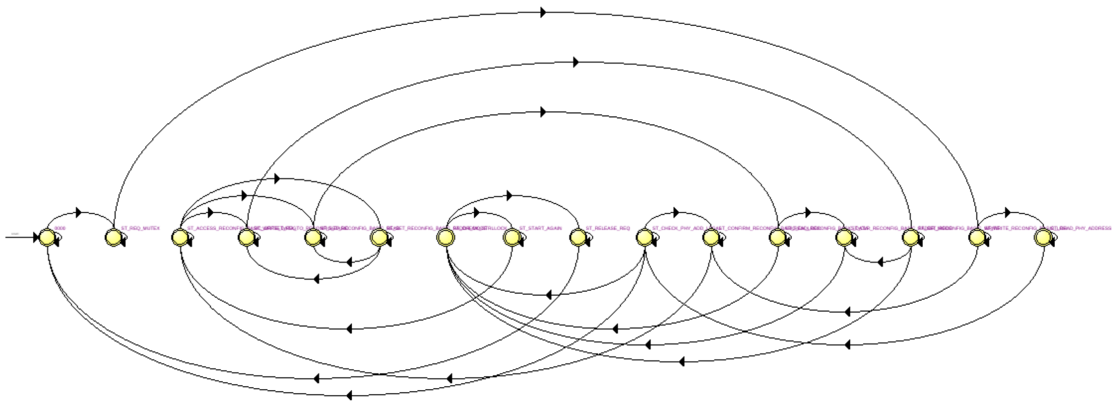
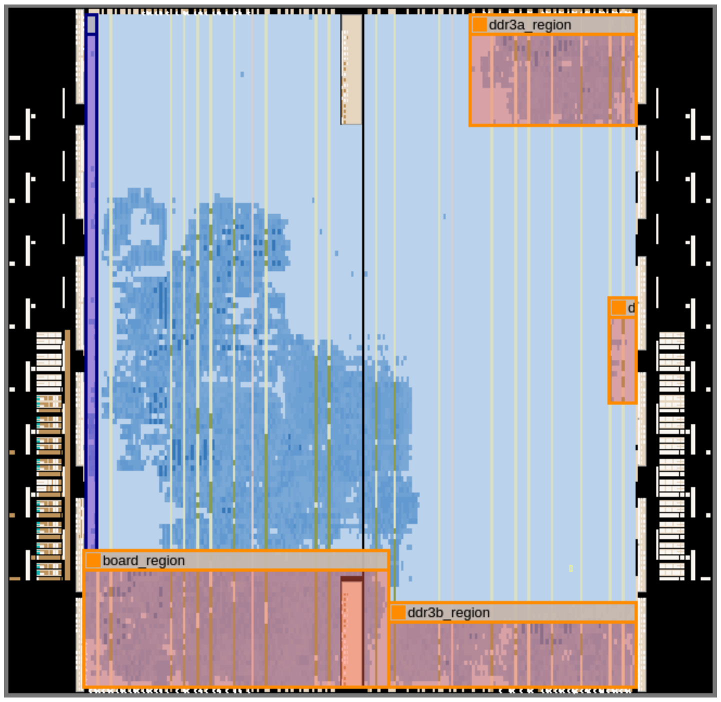
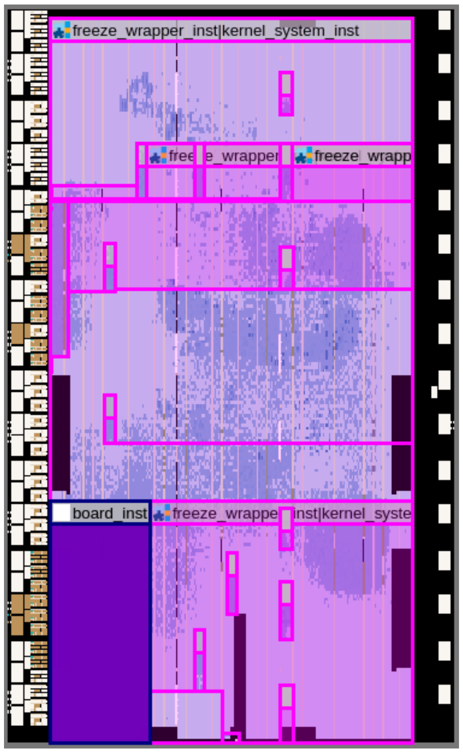


Figure 5. 3 State-Machine view of SOM FPGA design.

The chip planner feature of the Intel Quartus Prime Pro software was used to further analyze the design for resource and routing utilization and to view the power map of the FPGA design. Chip planner provides a visual display of the device resources, illustrating the arrangement of the resource atoms (i.e. Arithmetic Logic Modules (ALMs), Phase-Lock Loops (PLLs), DSP blocks, Memory blocks and I/O elements) in the device architecture. Fig. 5.4 shows the chip view of our SOM design on both Stratix V and Arria 10 FPGAs. The blocks labelled as *board_region, ddr_region, board_inst* and *freezer_wrapper_inst* denotes logic lock regions. The light green (i.e. vertical lines) represent the memory cells and the blue cells represents the Logic Blocks (LBs). Each LB consists of 16 individual Logic Elements (LEs). LBs utilized by our SOM design are indicated by a deeper blue shade.



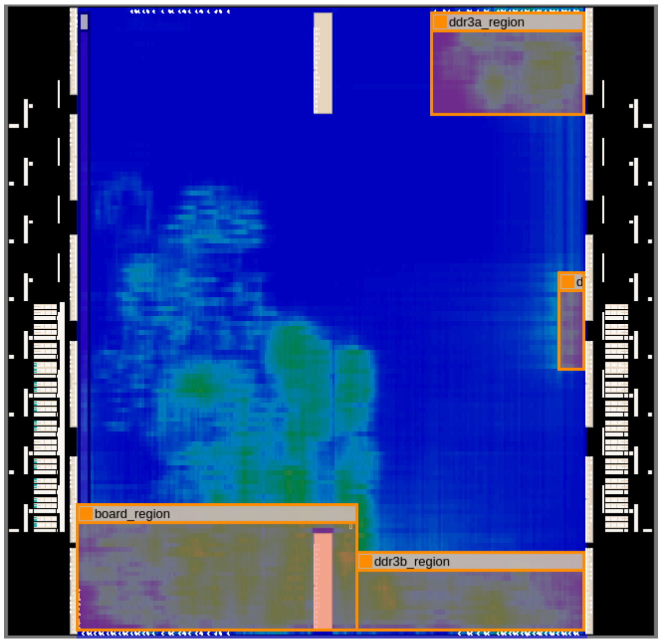
(a)


freeze_wrapper_inst|kernel_system_inst
freeze_wrapper
freeze_wrapp
board_inst
freeze_wrapper_inst|kernel_syste

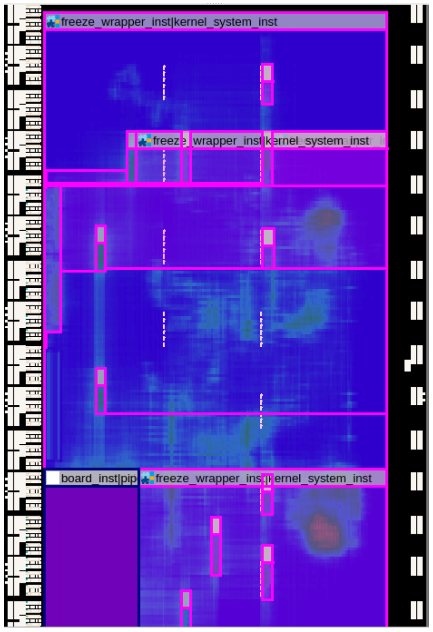
(b)

Figure 5. 4 Chip view of SOM design (a) Stratix V and (b) Arria 10.

Fig. 5.5 gives a view of the routing congestion of the SOM design for both Stratix V and Arria 10 FPGA obtained from the routing utilization feature of the software. Information from the feature can be used to ease routing congestion. Threshold value indicating the area of the chip considered as a high congestion area for the SOM design was set to 95%. Routing utilization as seen in Fig. 5.5, is displayed as a heat map of the logic resources, indicating relative resource utilization. Greater utilization is represented by hotter colors such as red/yellow and lower or zero utilization is represented by cooler colors such as green/blue.



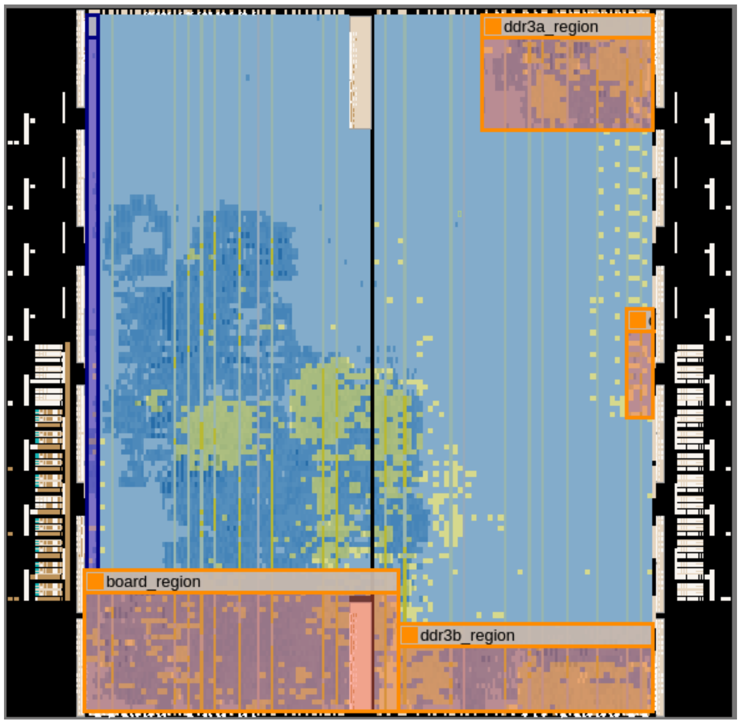
(a)


freeze_wrapper_inst|kernel_system_inst
freeze_wrapper_inst|kernel_system_inst
board_inst|pip
freeze_wrapper_inst|kernel_system_inst

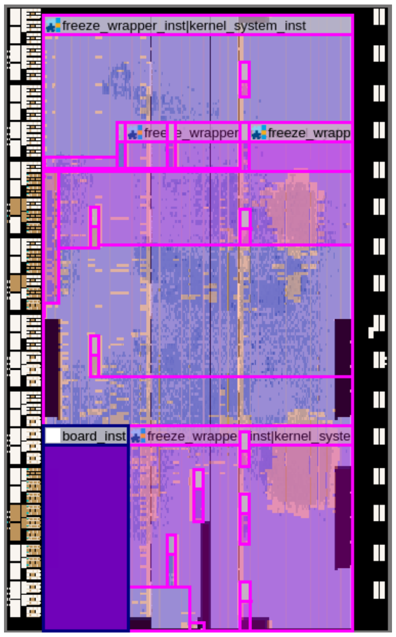
(b)

Figure 5. 5 Chip view indicating routing congestion of SOM design (a) Stratix V and (b) Arria 10.

Fig. 5.6 represents the power map of the SOM FPGA design indicating the High-Speed/Low-Power Tiles consisting of ALMs for both Stratix V and Arria 10 FPGAs. The tiles are differentiated by contrasting colors, where the yellow color represents High-Speed Tiles and deep blue color represents Low-Power Tiles.



(a)


freeze_wrapper_inst|kernel_system_inst
freeze_wrapper
freeze_wrapp
board_inst
freeze_wrapper_inst|kernel_syste

(b)

Figure 5. 6 Power Map of SOM design (a) Stratix V and (b) Arria 10.

5.5 Performance Analysis

The datasets were tested by launching the host program with different map sizes, input sizes and dimensions separately for both FPGA and GPU implementations. Automated test scripts were used for running the program and for generating and comparing experimental results in this research. The performance for different tests is measured by execution time in seconds (s), throughput in Floating Point Operations Per Second (FLOPS) and power in Watts (W).

5.5.1 Implementation in CPU

The SOM algorithm was implemented in AMD Athlon II 170u Processor @2.0GHz CPU. Table 5.1. shows the execution time for the SOM-CPU algorithm implementation for different map and input sizes. Table 5.2. shows the throughput for the SOM-CPU algorithm implementation for different map and input sizes. Table 5.3. shows execution time for the SOM-CPU algorithm implementation for a map size of 16x16 with 5120 inputs and varying dimensions.

Table 5. 1 SOM-CPU execution time for different map and input sizes

Input size	Execution time (s) for different Map Sizes				
	8x8	12x12	16x16	20x20	24x24
1024	5.02	16.32	35.50	71.43	122.92
2048	9.24	31.39	69.47	140.35	242.64
3072	13.62	46.09	120.60	209.66	396.08
4096	17.81	60.96	196.44	278.91	485.25
5120	22.02	76.02	224.21	349.11	603.87

Table 5. 2 SOM-CPU throughput for different map and input sizes

Input size	Throughput (FLOPS) for different Map Sizes				
	8x8	12x12	16x16	20x20	24x24
1024	783298.80	542283.79	443097.73	344061.94	287908.62
2048	851209.01	563669.72	452792.12	350212.68	291698.46
3072	866368.98	575825.05	391249.97	351655.06	268046.99
4096	883333.71	580506.86	320273.67	352457.78	291723.10
5120	893063.82	581910.02	350753.75	351983.78	293021.04
Average Throughput	855454.86	568839.09	391633.45	350074.25	286479.64

Table 5. 3 SOM-CPU Implementation for different dimensions

Dimension	Execution Time (s)
3	224.21
4	213.77
5	264.98
6	321.06

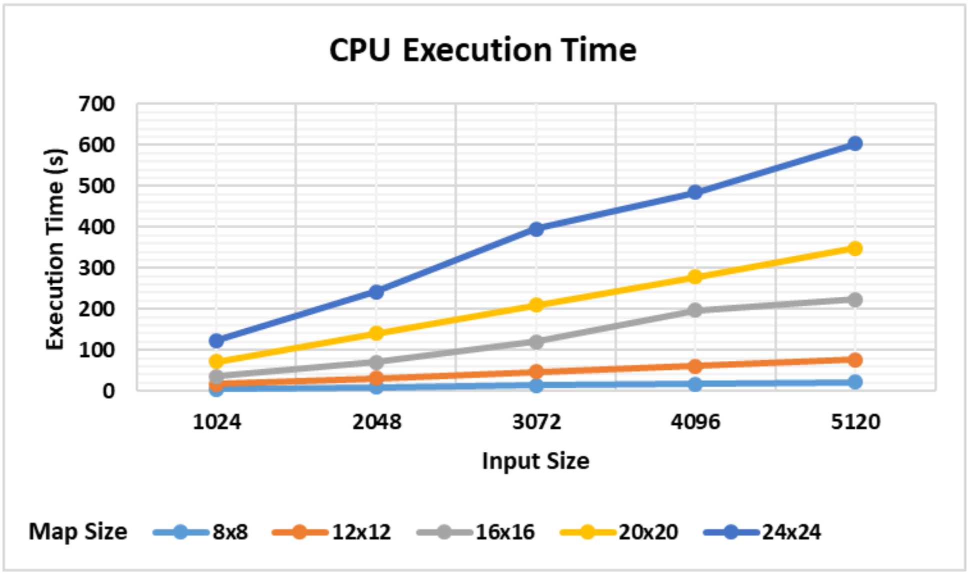


Figure 5. 7 Raw execution time for SOM on AMD Athlon II CPU for different map and input sizes.

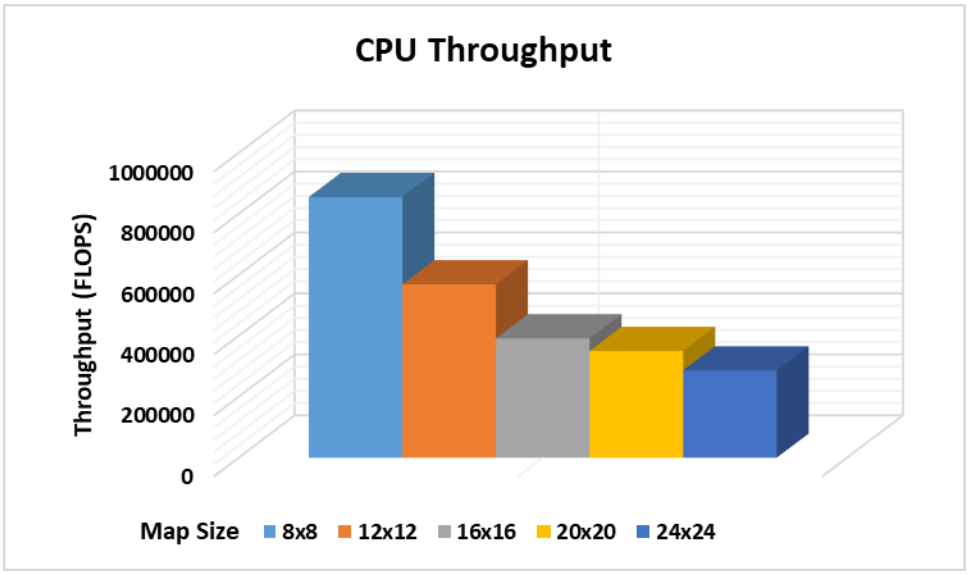


Figure 5. 8 Raw throughput for SOM on AMD Athlon II CPU for different map and input sizes.

The execution result of the SOM algorithm in CPU are shown in Fig 5.7 in terms of execution time (s) and Fig. 5.8 in terms of throughput (FLOPS). It can be observed that as the map size and input size increases, the execution time increases and the throughput decreases.

5.5.2 Implementation in GPU

The SOM algorithm was implemented in CUDA on Nvidia Quadro K620 GPU using the same operational flow and parameters used in FPGA implementation discussed in Chapter 4 for the purpose of comparison with FPGA implementation proposed in this research. Table 5.4 shows the execution time for the SOM-GPU algorithm implementation for different map and input sizes. Table 5.5. shows the throughput for the SOM-GPU algorithm implementation for different map and input sizes. Table 5.6. shows execution time for the SOM-GPU algorithm implementation for a map size of 16x16 with 5120 inputs and varying dimensions.

Table 5. 4 SOM-GPU execution time for different map and input sizes

Input size	Execution time (s) for different Map Sizes				
	8x8	12x12	16x16	20x20	24x24
1024	8.75	31.99	69.25	149.04	257.73
2048	17.49	63.96	138.49	298.18	515.55
3072	26.23	95.96	207.73	447.31	773.31
4096	34.97	127.96	276.97	596.34	1031.07
5120	43.73	159.98	346.32	745.43	1288.93

Table 5. 5 SOM-GPU throughput for different map and input sizes

Input size	Throughput (FLOPS) for different Map Sizes				
	8x8	12x12	16x16	20x20	24x24
1024	449543.84	276601.01	227131.66	164893.12	137311.54
2048	449723.80	276652.91	227149.70	164839.48	137289.43
3072	449766.66	276598.13	227151.34	164823.45	137290.76
4096	449788.10	276557.78	227149.70	164845.56	137292.10
5120	449646.66	276521.48	227081.81	164844.67	137281.92
Average Throughput	449693.81	276586.26	227132.8392	164849.25	137293.15

Table 5. 6 SOM-GPU implementation for different dimensions

Dimension	Execution Time (s)
3	346.32
4	322.78
5	435.86
6	486.87

The execution result of the SOM algorithm in GPU are shown in Fig 5.9 in terms of execution time (s) and Fig. 5.10 in terms of throughput (FLOPS). It can be observed that as the map size and input size increases, the execution time increases and the throughput decreases.

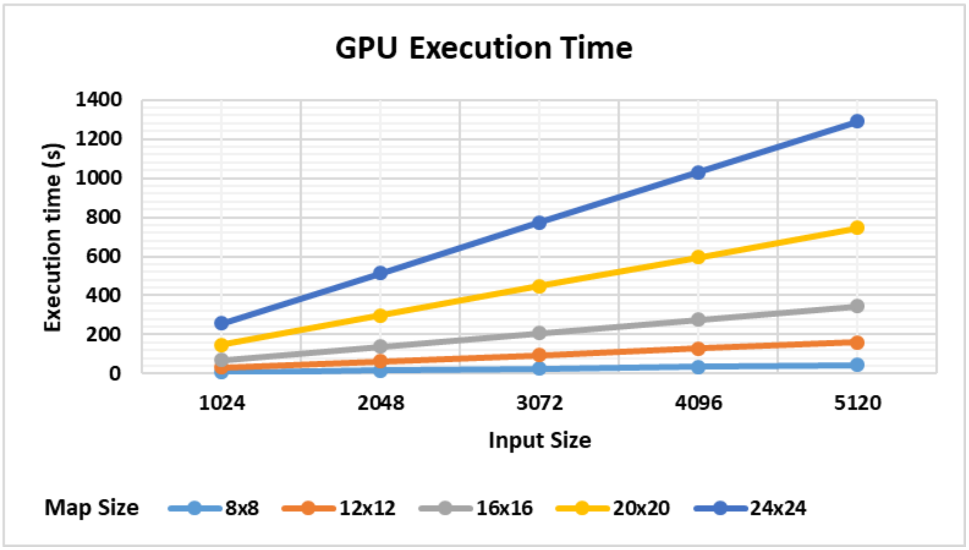


Figure 5. 9 Raw execution time for SOM on Nvidia Quadro K620 GPU for different map and input sizes.

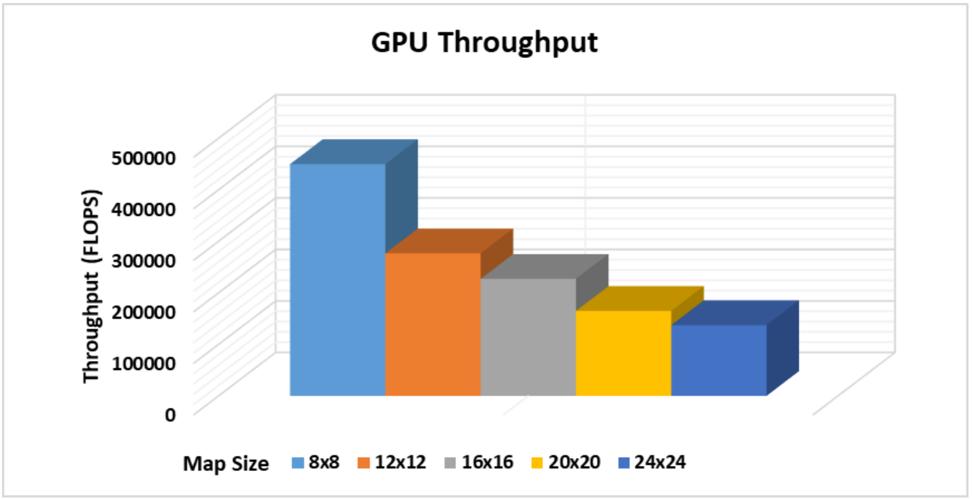


Figure 5. 10 Raw throughput for SOM on Nvidia Quadro K620 GPU for different map and input sizes.

5.5.3 Implementation in FPGA

The SOM algorithm was implemented in Intel FPGA SDK for OpenCL on Stratix V and Arria 10 FPGA accelerator boards. Table 5.7 and Table 5.8, shows the execution time for the SOM-FPGA algorithm implementation on Stratix V and Arria 10 FPGA for different map and input sizes. Table 5.9 and Table 5.10, shows the throughput for the SOM-FPGA algorithm implementation on Stratix V and Arria 10 FPGA for different map and input sizes. Table 5.11 and Table 5.12, shows execution time for the SOM-FPGA algorithm implementation on Stratix V and Arria 10 FPGA for a map size of 16x16 with 5120 inputs and varying dimensions. The execution result for the SOM algorithm in FPGA are shown in Fig 5.11 and Fig. 5.12 in terms of execution time (s) and Fig. 5.13 and Fig. 5.14 in terms of throughput (FLOPS). It can be observed that as the map size and input size increases, the execution time increases and the throughput decreases.

Table 5. 7 SOM-FPGA execution time on Stratix V FPGA for different map and input sizes

Input size	Execution time (s) for different Map Sizes				
	8x8	12x12	16x16	20x20	24x24
1024	3.74	15.38	29.18	55.29	97.56
2048	7.44	33.62	58.80	110.25	210.04
3072	12.35	40.78	98.50	188.83	330.19
4096	16.34	54.60	128.71	251.76	436.99
5120	19.51	64.68	159.22	314.68	531.06

Table 5. 8 SOM-FPGA execution time on Arria 10 FPGA for different map and input sizes

Input size	Execution time (s) for different Map Sizes				
	8x8	12x12	16x16	20x20	24x24
1024	3.43	12.38	28.13	53.77	87.33
2048	7.55	32.58	54.07	107.10	185.25
3072	10.37	34.70	81.57	159.74	288.66
4096	14.27	46.59	107.55	214.65	384.14
5120	17.97	56.56	137.63	280.17	477.04

Table 5. 9 SOM-FPGA throughput on Stratix V FPGA for different map and input sizes

Input size	Throughput (FLOPS) for different Map Sizes				
	8x8	12x12	16x16	20x20	24x24
1024	1050219.94	575399.50	539116.03	444479.11	362740.28
2048	1057612.24	526383.08	534990.29	445831.09	336984.89
3072	955584.16	650858.79	479060.68	390442.40	321539.92
4096	962346.13	648118.12	488813.88	390469.28	323937.85
5120	1007934.99	683954.25	493934.88	390490.20	333194.28
Average Throughput	1006739.49	616942.75	507183.15	412342.42	335679.44

Table 5. 10 SOM-FPGA throughput on Arria 10 FPGA for different map and input sizes

Input size	Throughput (FLOPS) for different Map Sizes				
	8x8	12x12	16x16	20x20	24x24
1024	1148012.90	714396.11	559115.89	457022.57	405217.21
2048	1041028.96	543040.52	581822.31	458950.62	382067.60
3072	1137330.22	764883.45	578443.42	461558.66	367798.62
4096	1102357.37	759549.68	584962.40	457974.16	368504.48
5120	1094182.46	782075.46	571416.74	438588.87	370927.38
Average Throughput	1104582.38	712789.04	575152.15	454818.98	378903.06

Table 5. 11 SOM-FPGA implementation on Stratix V FPGA for different dimensions

Dimension	Execution Time (s)
3	159.22
4	97.03
5	170.71
6	198.52

Table 5. 12 SOM-FPGA implementation on Arria 10 FPGA for different dimensions

Dimension	Execution Time (s)
3	137.63
4	75.72
5	152.03
6	147.02

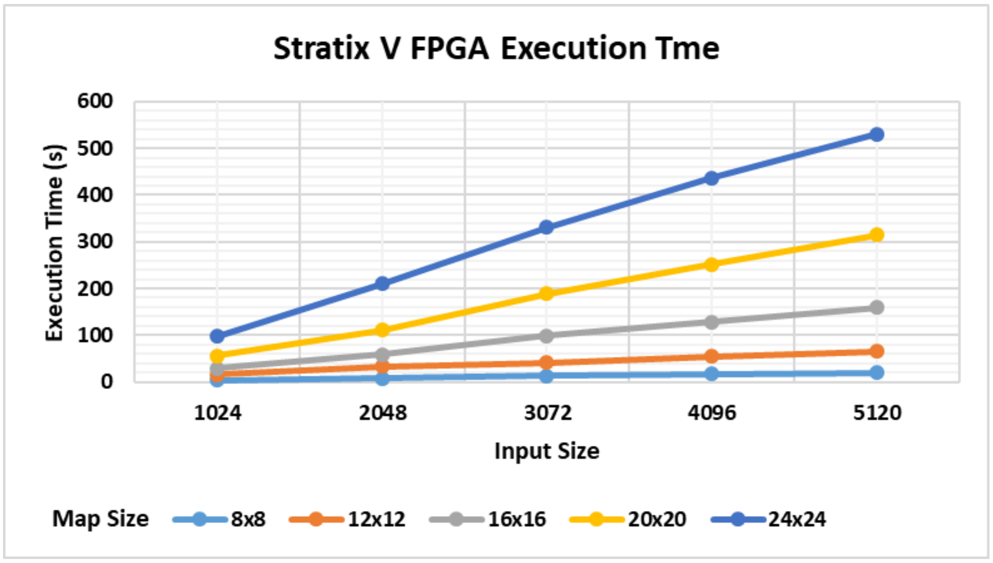


Figure 5. 11 Raw execution time for SOM on Stratix V FPGA for different map and input sizes.

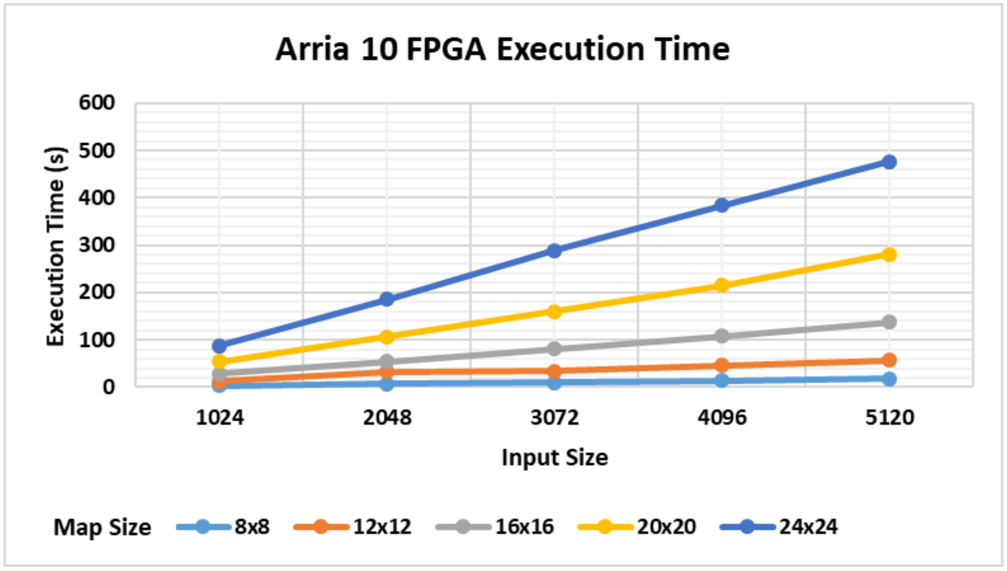


Figure 5. 12 Raw execution time for SOM on Arria 10 FPGA for different map and input sizes.

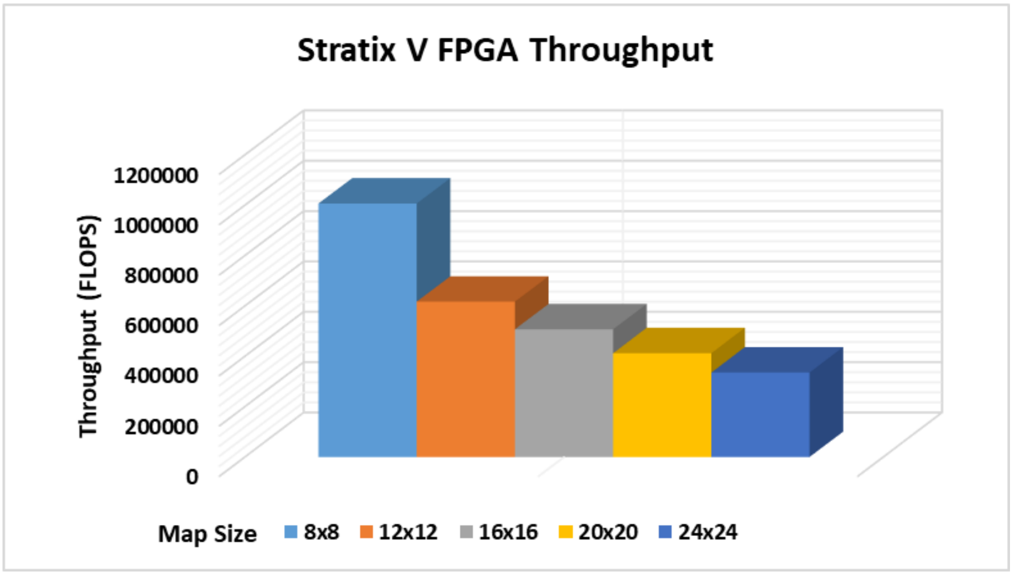


Figure 5. 13 Raw throughput for SOM on Stratix V FPGA for different map and input sizes.

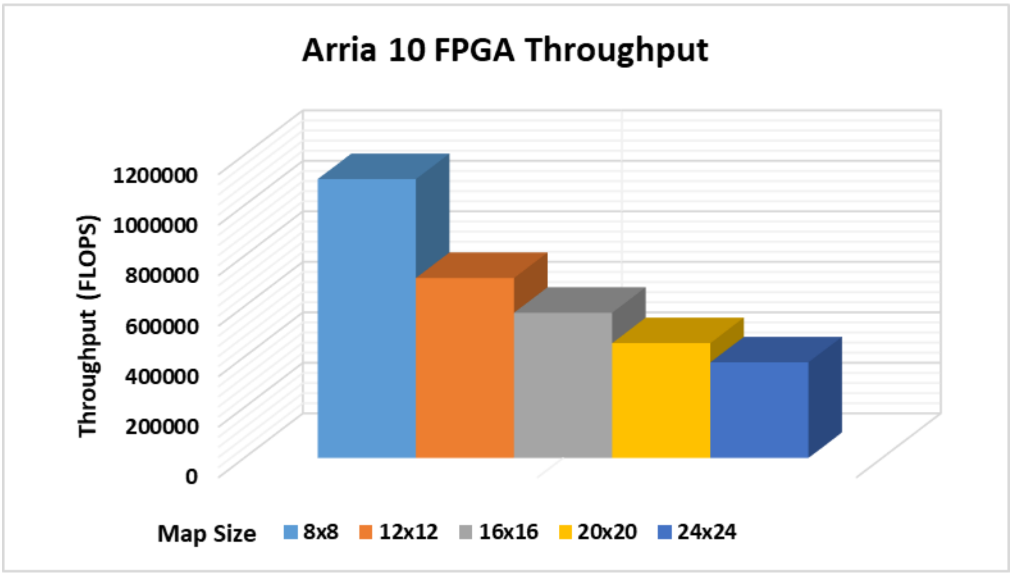


Figure 5. 14 Raw throughput for SOM on Arria 10 FPGA for different map and input sizes.

5.5.4 Performance Comparison between CPU and FPGA

Fig. 5.15 and Fig. 5.16, shows the execution time and throughput (FLOPS) comparison for different map sizes and dimensions for AMD Athlon II CPU, Stratix V FPGA and Arria 10 FPGA respectively. It is observed that for all map sizes Arria 10 FPGA gives the highest throughput followed by Stratix V FPGA, AMD Athlon II CPU gives the lowest throughput in all cases compared to the FPGA devices. The implementation in Fig 5.16 by varying dimensions was done for map size of 16x16 – 256 neurons with input size of 5120 points.

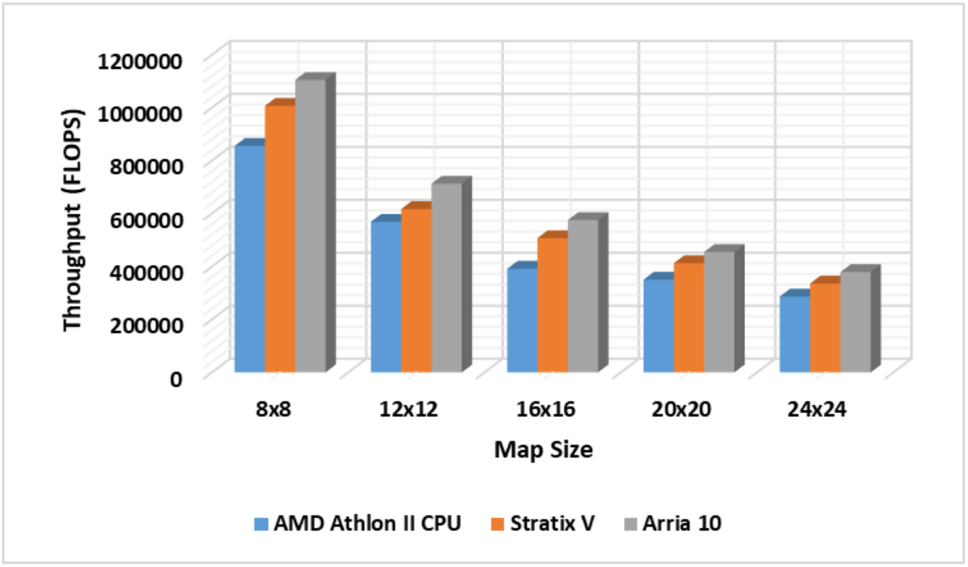


Figure 5. 15 Comparison of throughput for SOM implementation on AMD Athlon II CPU, Stratix V and Arria 10 FPGAs for different map and input sizes.

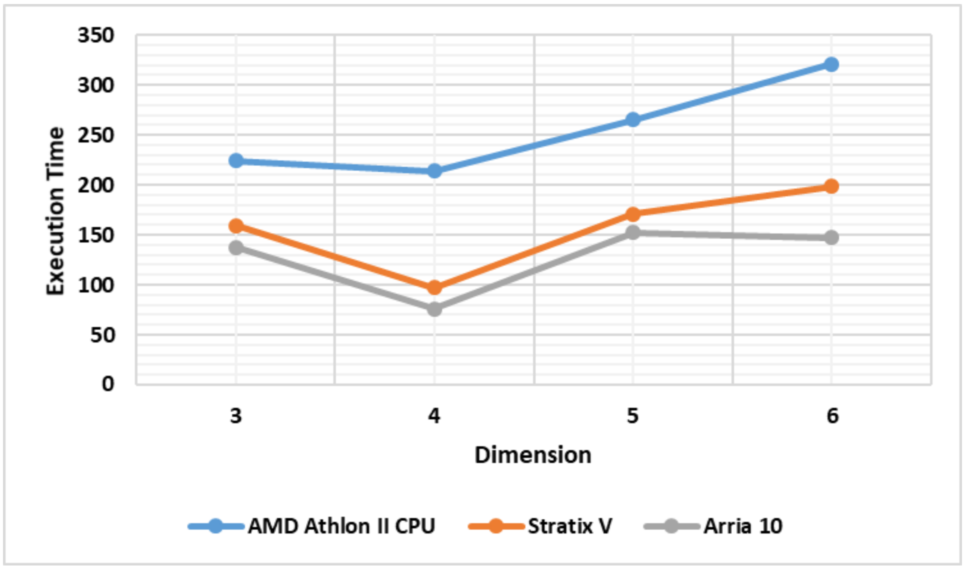


Figure 5. 16 Comparison of throughput for SOM implementation on AMD Athlon II CPU, Stratix V and Arria 10 FPGAs for different dimensions.

For the implementation shown in Fig 5.17 and Fig. 5.18, we chose map size: 16x16 – 256 neurons, input size: 5120 points and dimension: 3. The CPU (AMD Operton 6366HE and Intel core i7-2600) implementation result were obtained from [7] for comparison with our FPGA (Stratix V and Arria 10 using OpenCL) and CPU (AMD Athlon II) implementation. From Fig. 5.17 and Fig. 5.18, it can be observed that SOM-CPU has higher execution time and lower throughput compared to SOM-FPGAs. From Fig. 5.19, it can be also concluded that Stratix V achieved 16.55x, 2.53x and 1.41x speedup compared to AMD Operton 6366HE and Intel core i7-2600 CPU [7] and our AMD Athlon II CPU implementation and Arria 10 achieved 19.15x, 2.93x and 1.63x speedup compared to AMD Operton 6366HE and Intel core i7-2600 CPU [7] and our AMD Athlon II CPU implementation respectively.

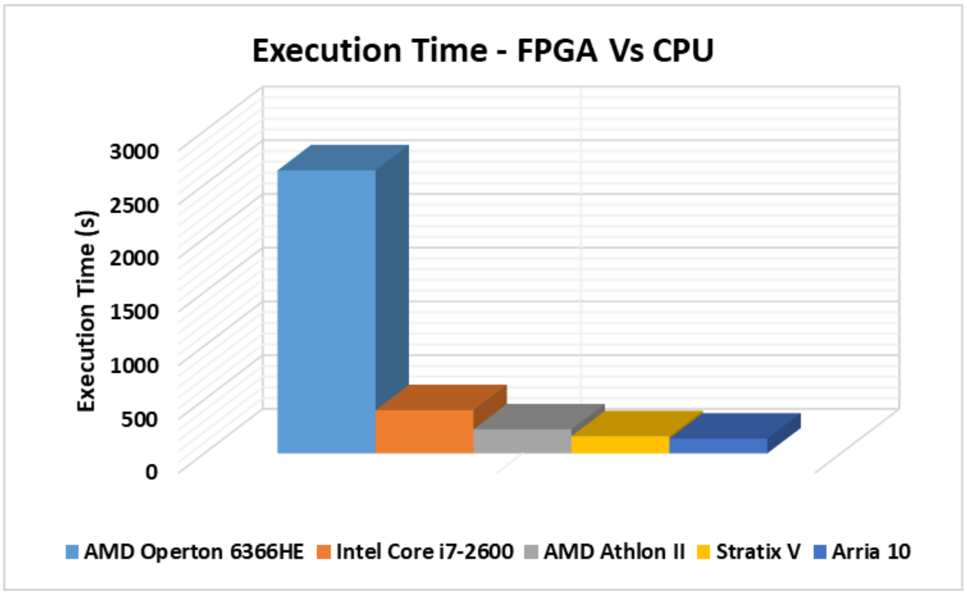


Figure 5. 17 Comparison of execution time for SOM implementation between CPU and FPGA.

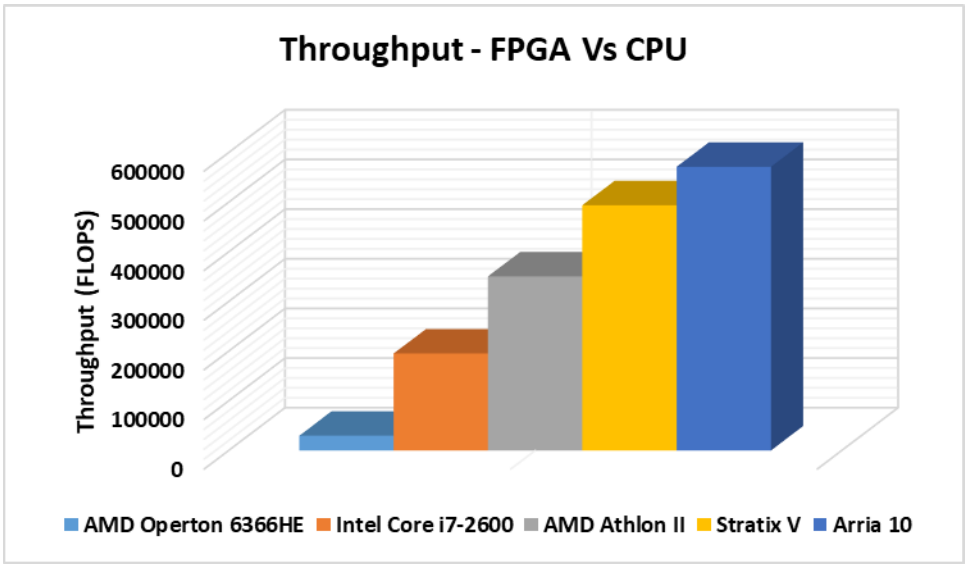
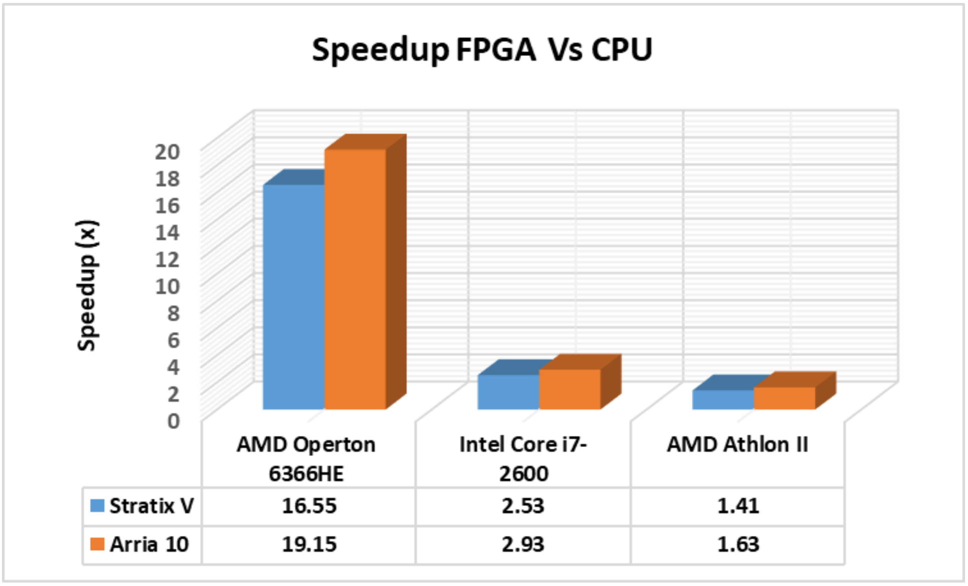


Figure 5. 18 Comparison of throughput for SOM implementation between CPU and FPGA.



	AMD Operton 6366HE	Intel Core i7-2600	AMD Athlon II
🟦 Stratix V	16.55	2.53	1.41
🟧 Arria 10	19.15	2.93	1.63

Figure 5. 19 Speedup comparison between FPGA and CPU.

5.5.5 Performance Comparison between GPU and FPGA

Fig 5.20 and Fig. 5.21 shows the execution time and throughput (FLOPS) comparison for different map sizes and dimensions for Nvidia GeForce GTX 590 [7], Nvidia Quadro K620 GPU, Stratix V FPGA and Arria 10 FPGA respectively. For comparison between FPGA and GPU, as shown in Fig 5.20 and Fig 5.21, we chose map size: 16x16 – 256 neurons, input size: 5120 points and dimension: 3. The GPU (Nvidia GeForce GTX 590 using OpenCL) implementation result were obtained from [7] for comparison with our FPGA (Stratix V and Arria 10 using OpenCL) and GPU (Nvidia Quadro K620 using CUDA) implementation. From the implementations, as shown in Fig. 21 and Fig. 22, it can be concluded that Stratix V and Arria 10 achieved speedup of 2.18x and 2.52x compared to Nvidia Quadro K620 GPU and is slightly slower or similar in terms of both execution time and throughput with that published in [20] using GPU. The implementation in Fig 5.21 by varying dimensions was done for map size of 16x16 – 256 neurons with input size of 5120 points.

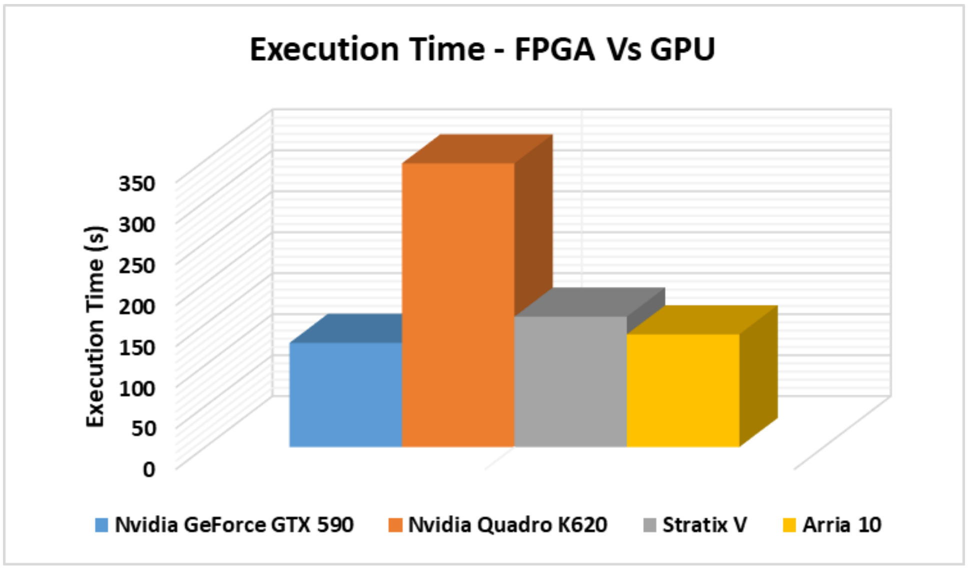


Figure 5. 20 Comparison of execution time for SOM implementation between GPU and FPGA.

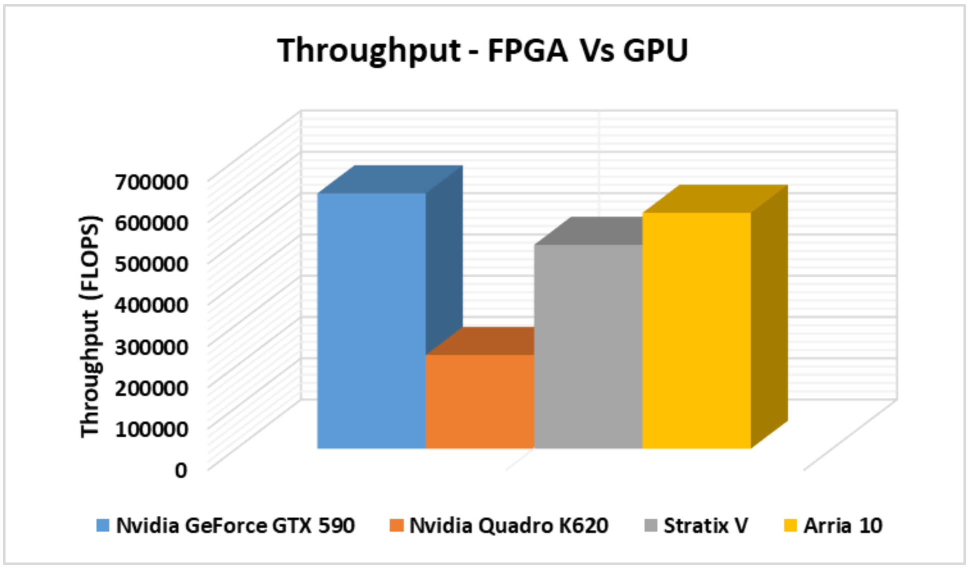
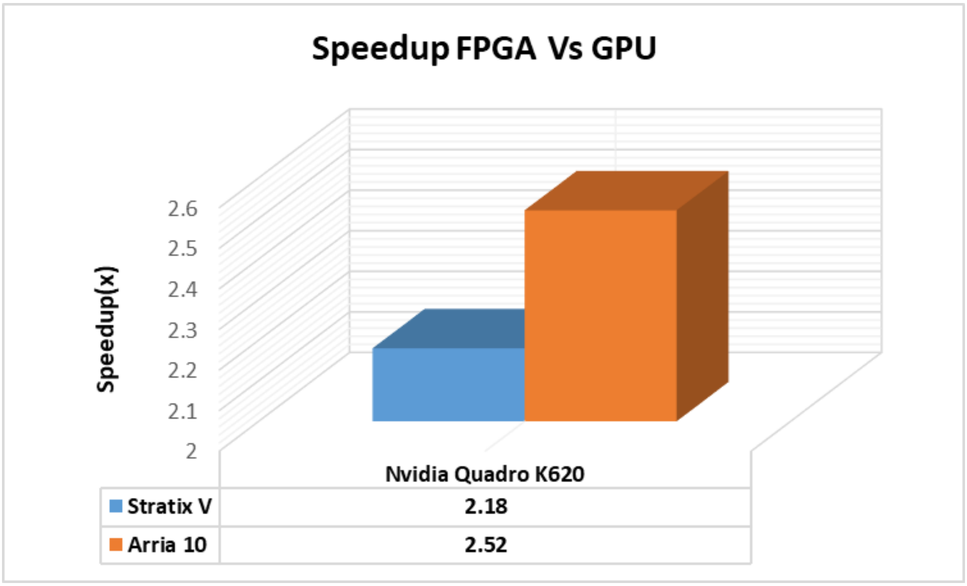


Figure 5. 21 Comparison of throughput for SOM implementation between GPU and FPGA.



	Nvidia Quadro K620
🟦 Stratix V	2.18
🟧 Arria 10	2.52

Figure 5. 22 Speedup comparison between FPGA and GPU

5.5.6 Performance Comparison between FPGAs

Fig. 5.23 and Fig 5.24, shows the speedup comparison between Stratix V and Arria 10 FPGAs in terms of throughput (FLOPS) for different map sizes and dimensions. The implementation in Fig 5.23 by varying dimensions was done for map size of 16x16 – 256 neurons with input size of 5120 points. From this implementation, it can be concluded that Arria 10 FPGA shows better performance and has achieved a speedup of 1.12x compared to Stratix V FPGA.

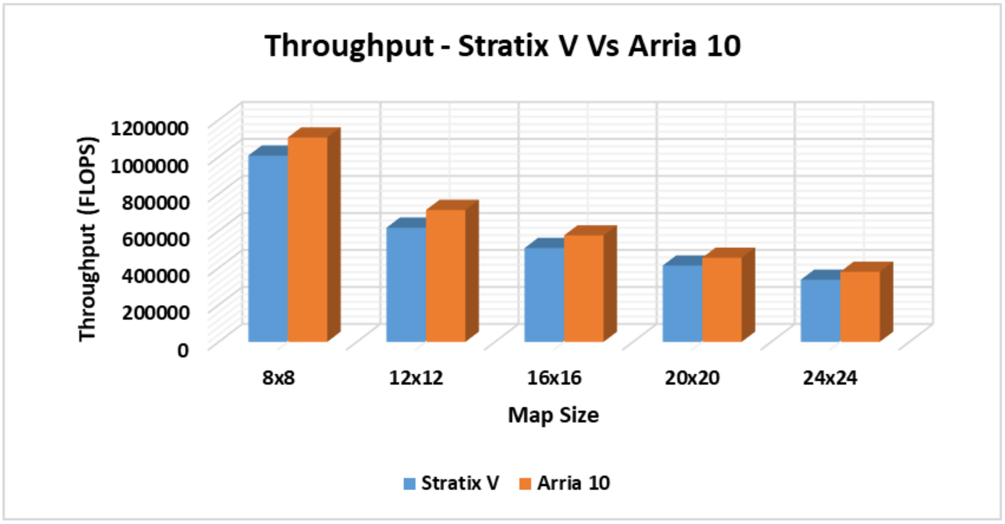


Figure 5. 23 Comparison of throughput for SOM implementation between Stratix V and Arria 10 FPGAs for different map sizes.

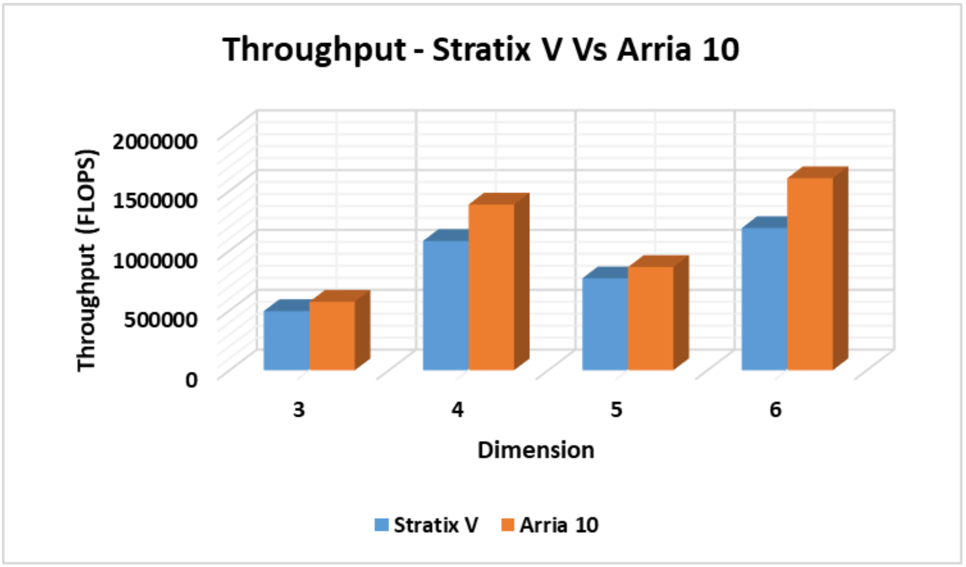


Figure 5. 24 Comparison of throughput for SOM implementation between Stratix V and Arria 10 FPGAs for different dimensions.

5.6 Resource Utilization

Evaluation of resource utilized by the kernels are conducted by compiling the kernel using the AOC compiler for different data features such as map and input sizes and dimensions. The clock frequency (Kernel f_{max}) and hardware utilization such as Logic Utilization, Adaptive Look-Up Tables (ALUTs), Dedicated Logic Registers, Memory Blocks and Digital Signal Processing Blocks are different for each implementation. The resource utilization for different map sizes is shown in Table 5.13, Table 5.14 and Fig. 5.25, Fig. 5.26. The clock frequency in MHz for different implementations shown in Fig. 5.27 and Fig 5.28, is dependent upon the complexity of the HDL design generated by the AOCL. The operating frequency drops as the design becomes more and more complex and thus the latency of computation increases. Table 5.15, Table 5.16 and Fig. 5.29, Fig. 5.30, shows the resource utilization for different dimensions for a map size of 16x16 - 256 neurons and input size of 5120. For our implementation, the resource utilization for different map and input sizes and dimensions was well below 45%, which indicates that maps and input sizes and dimensions of higher values can be implemented on the FPGAs before the resource utilization reaches a limit, which makes it difficult for the Quartus software to fit the design on the FPGA. To better fit the design on the FPGA decreasing/removing loop unroll factor for some and/or all loops inside each kernel was implemented. For this reason, for some implementations, FPGA resource usage and operating frequency dropped down compared to other implementations.

Table 5. 13 Stratix V FPGA resource utilization for different map sizes

Map Size	Logic Utilization	ALUTs	Dedicated Logic Registers	Memory Blocks	DSP Blocks
8x8	28	17	12	26	5
12x12	30	18	13	29	7
16x16	30	17	14	27	5
20x20	31	19	14	33	7
24x24	32	19	14	32	7

Table 5. 14 Arria 10 FPGA resource utilization for different map sizes

Map Size	Logic Utilization	ALUTs	Dedicated Logic Registers	Memory Blocks	DSP Blocks
8x8	39	15	24	22	6
12x12	40	16	25	24	7
16x16	40	15	25	23	6
20x20	41	16	25	29	7
24x24	44	16	25	27	7

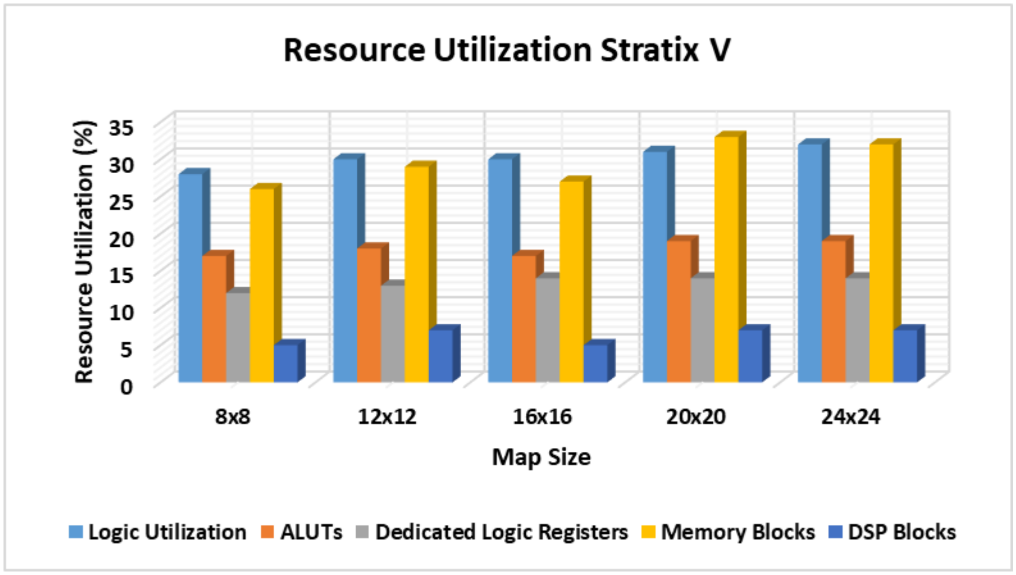


Figure 5. 25 Stratix V FPGA resource utilization for different map sizes.

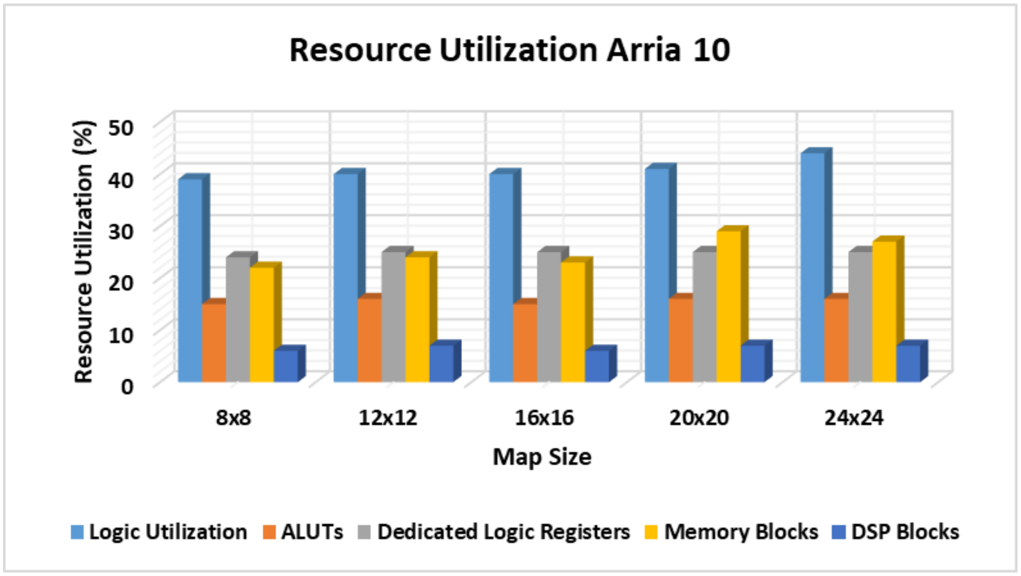


Figure 5. 26 Arria 10 FPGA resource utilization for different map sizes.

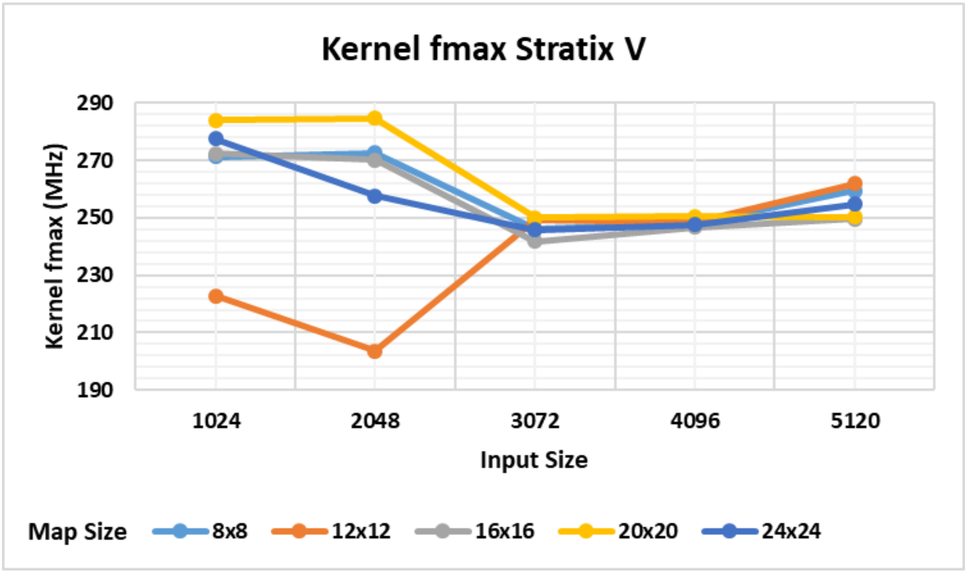


Figure 5. 27 Stratix V FPGA clock frequency kernel (f_{max}) for different input sizes.

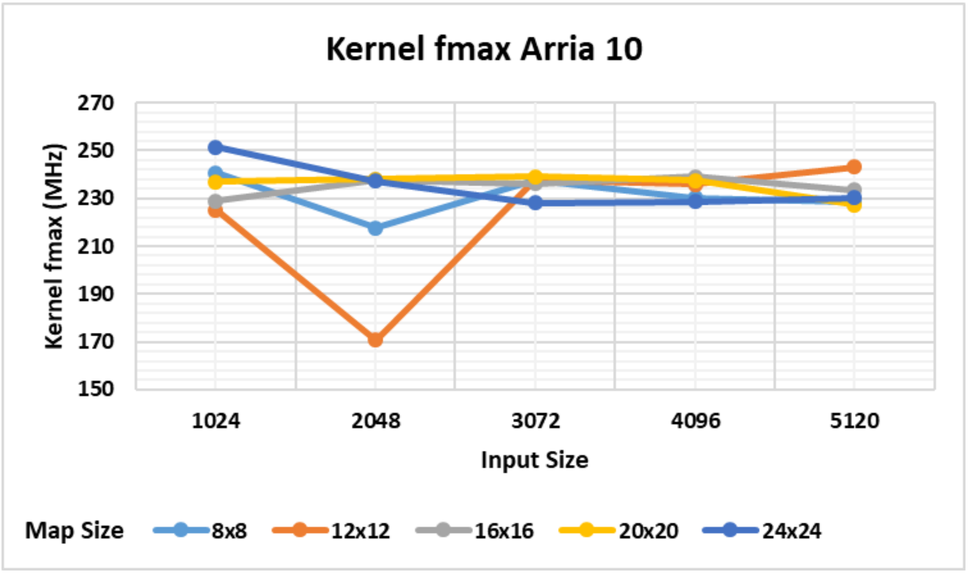


Figure 5. 28 Arria 10 FPGA clock frequency kernel (f_{max}) for different input sizes.

Table 5. 15 Stratix V FPGA resource utilization for different dimensions

Map Size	Logic Utilization	ALUTs	Dedicated Logic Registers	Memory Blocks	DSP Blocks
8x8	30	17	14	27	5
12x12	28	17	13	25	0
16x16	31	18	14	31	5
20x20	31	18	14	31	5
24x24	30	17	14	27	5

Table 5. 16 Arria 10 FPGA resource utilization for different dimensions

Map Size	Logic Utilization	ALUTs	Dedicated Logic Registers	Memory Blocks	DSP Blocks
8x8	40	15	25	23	6
12x12	39	15	24	21	6
16x16	41	16	25	27	7
20x20	41	16	25	27	7
24x24	40	15	25	23	6

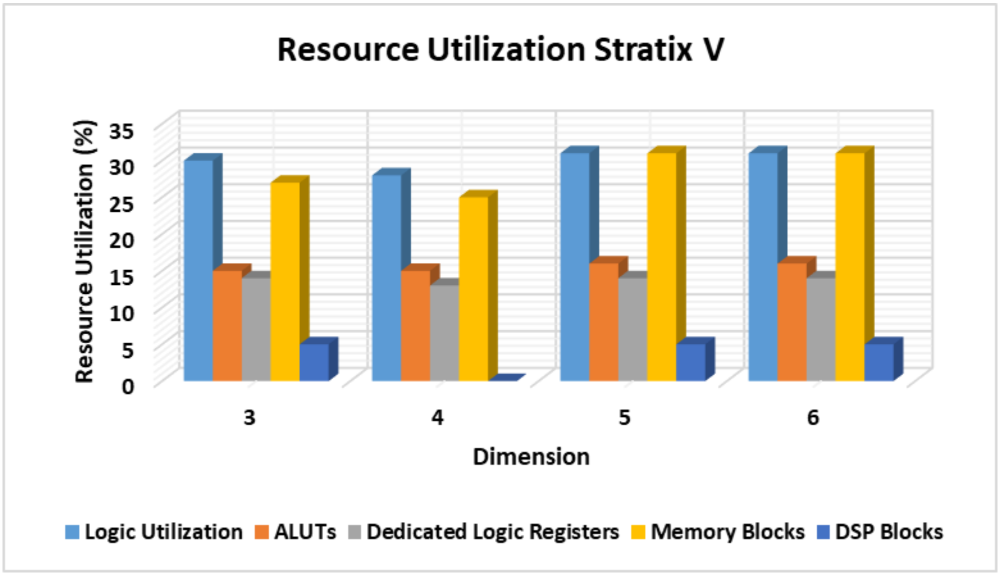


Figure 5. 29 Stratix V FPGA resource utilization for different dimensions.

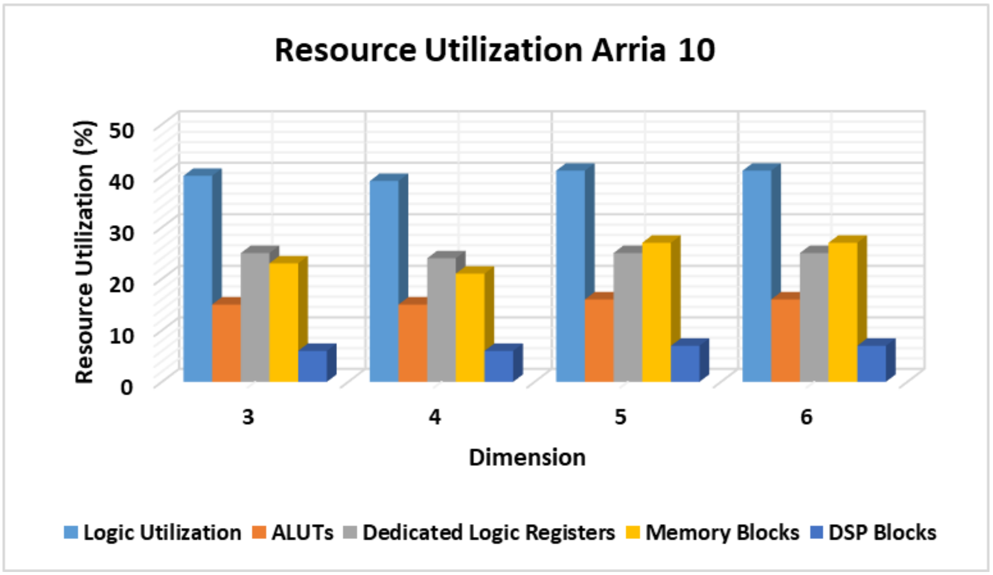


Figure 5. 30 Arria 10 FPGA resource utilization for different dimensions.

5.7 Energy Efficiency

In order to calculate the power consumption by various HPC platforms implementing the SOM algorithm we used “Watts up? Pro” power meter [65], which can be used to obtain true power consumed by the device with an accuracy of 1.5%. Table 5.17 shows the power consumption for two CPUs and FPGAs and GPU during the idle mode and program execution mode. The idle power corresponds to the power of the workstations when no computational tasks are assigned to them. The idle + execution power corresponds to the power obtained during the SOM algorithm execution in the respective devices. The actual execution power of the SOM algorithm was obtained by subtracting the Idle Power from the (Idle + Execution Power).

Table 5. 17 Power Consumption of CPUs, GPU and FPGAs.

System	CPU with Stratix V Board		CPU with Arria 10 Board		CPU with Nvidia Quadro K620 Board	
	CPU only	CPU with Board	CPU only	CPU with Board	CPU only	CPU with Board
Idle Power (W)	76.80	76.80	139.60	139.60	76.80	76.80
Idle + Execution Power (W)	107.00	77.65	175.50	141.87	107.00	112.95
Actual Execution Power (W)	30.20	0.85	35.90	2.27	30.20	36.15

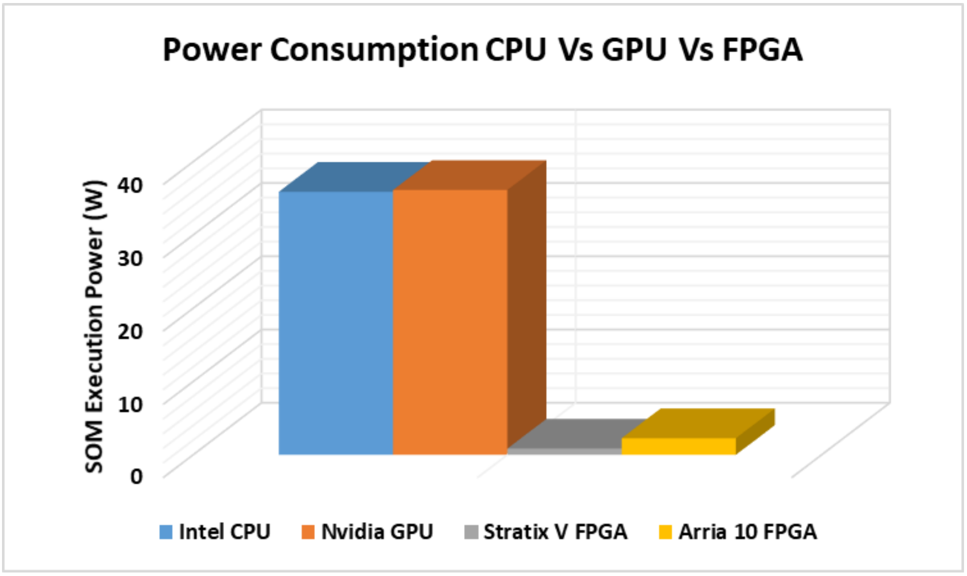


Figure 5. 31 SOM algorithm execution power for CPU, GPU and FPGAs.

Fig. 5.31 shows a comparison of the SOM execution power (W) between the CPU, GPU and FPGAs. From the power estimation as shown in Fig. 5.32 it was found out that Stratix V and Arria 10 are 35.53x and 15.82x more power efficient compared to CPU and 42.53x and 15.93x more power efficient compared to Nvidia Quadro K620 GPU. Moreover, it was found out that Stratix V was 2.67x more power efficient compared to Arria 10 for SOM implementation.

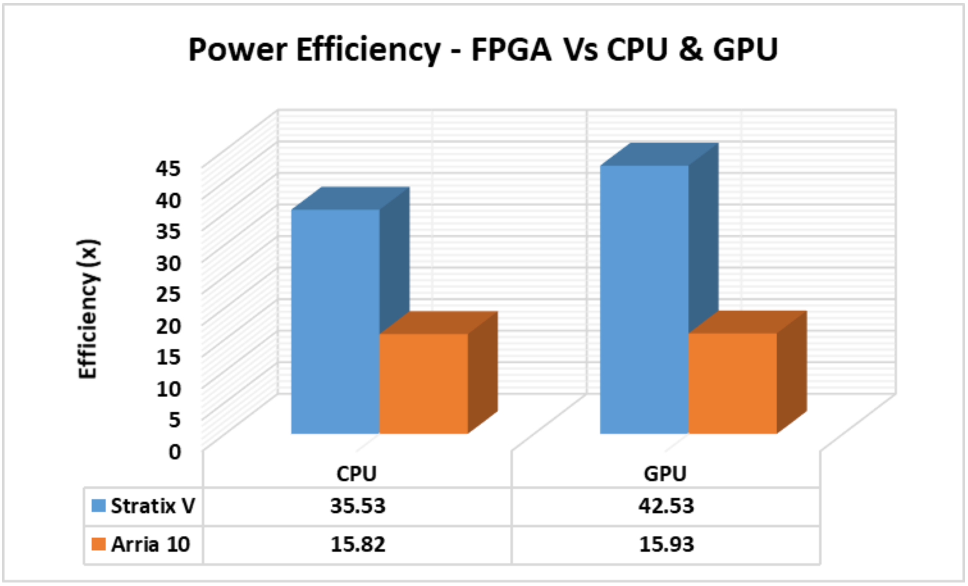


Figure 5. 32 Comparison of efficiency (i.e. in terms of power) obtained using FPGA compared to CPU and GPU.

5.8 Verification

A sequential version of SOM algorithm was implemented in CPU alongside FPGA and GPU in order to ensure the accuracy of our FPGA and GPU implementations. The implementation was done after SOM FPGA and GPU implementations respectively on the same dataset. After the execution of the kernels, the host enqueues read command in order to obtain the clustered data from the FPGA and GPU global memory to the host memory for verification with the CPU clustered data obtained after SOM CPU implementation. Two verification methods were used (1) Average Quantization Error and (2) Visual representation. The average quantization error is the comparison/mapping of how well the input values maps on to the output values. For implementations of varying map and input sizes and dimensions, the average quantization error for all implementations was found to be same for CPU, GPU and FPGA. It was found from the implementations that the value of average quantization error decreases as the map and input sizes and dimensions increased.

For the purpose of verification of our implementations, we employed a way to visually represent our resultant clustered dataset similar to the way implemented in [7 and 59]. Similar visual representations were obtained in case of CPU, GPU and FPGA for all data sets indicating the correctness our SOM implementation. Sample visual representation of the resultant clustered data obtained from CPU, GPU and FPGA is shown in Fig. 5.33.

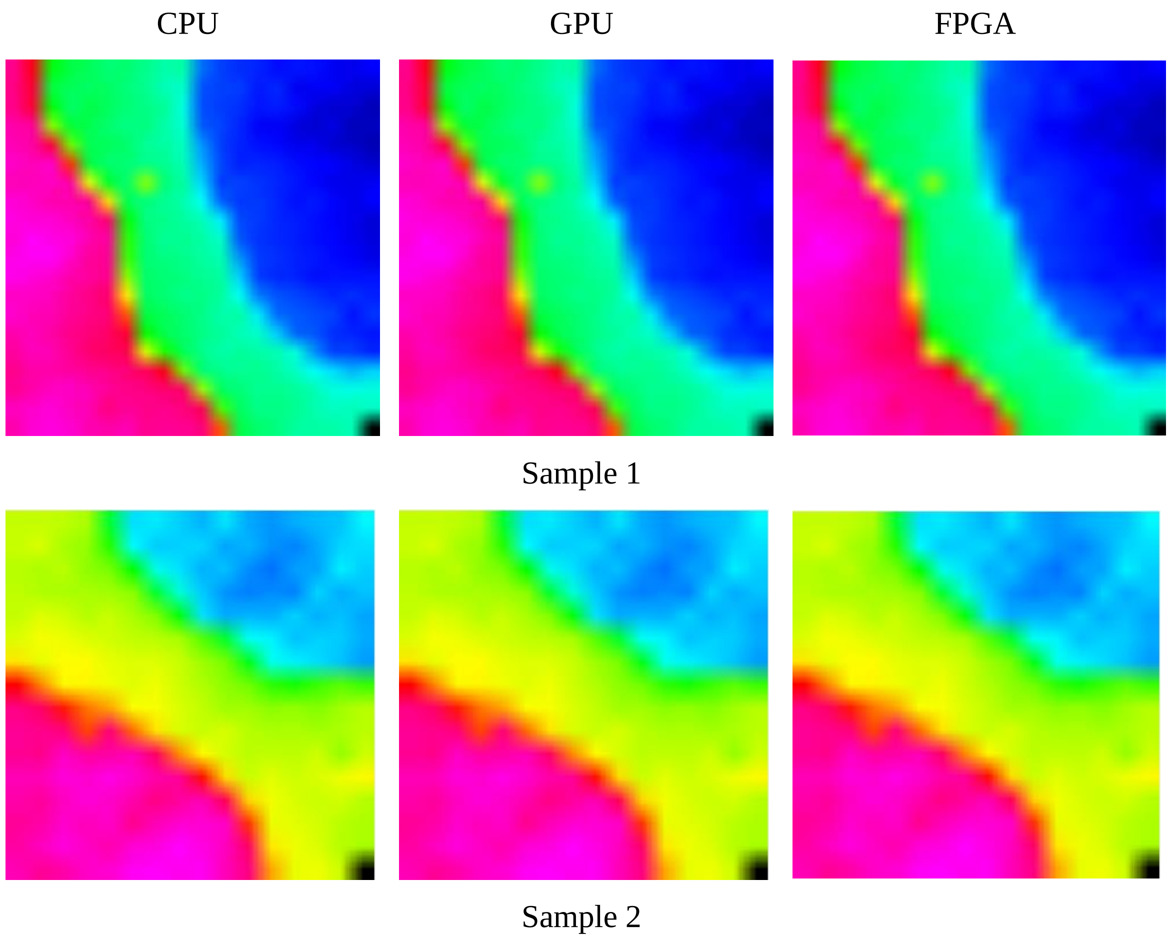


Figure 5. 33 Visual representation of SOM resultant clustered data obtained from HPC platforms (i.e. CPU/GPU/FPGA)

Chapter 6 Conclusion

An optimized FPGA based acceleration of SOM algorithm was implemented using IFSO. HLS implementation using Stratix V and Arria 10 FPGAs of SOM was evaluated against other HPC platforms such as CPUs and GPUs. We had to efficiently restructure the operational flow of the SOM algorithm to fully take advantage of the parallel nature of the algorithm.

Our FPGA implementation using Stratix V and Arria 10 was able to achieve speedup of 16.55x and 19.15x Vs AMD Operton CPU, 1.41x and 1.63x Vs AMD Athlon II CPU and 2.53x and 2.93x Vs Intel CPU respectively. Compared to Nvidia Quadro K620 GPU, our implementation using Stratix V and Arria 10 FPGA, achieved speedup of 2.18x and 2.52x respectively. Moreover, Arria 10 FPGA achieved speedup of 1.12x compared to Stratix V FPGA. In terms of power, Stratix V and Arria 10 are 35.53x and 15.82x more power efficient compared to CPU and 42.53x and 15.93x more power efficient compared to Nvidia Quadro K620 GPU.

Due to the recent advancement in FPGA technology and an increasing demand for FPGAs, it is likely that FPGAs (i.e. newer generations) could outperform GPUs in solving computationally intensive tasks. In this research, we used single chip FPGA based accelerators, Stratix V and Arria 10, it would be interesting to see how SOM algorithm performance and resource utilization would change when multi-chip FPGA accelerators and newer versions of the FPGA accelerator boards are used. Even though we were able to fit the design onto a single FPGA board, it would be interesting to see how the SOM algorithm performs compared to GPUs and CPUs when the same algorithm is implemented on multiple FPGA boards at the same time and the maximum number of neurons and dimensions can be used for SOM computation.

References

[1]H. Hikawa and Y. Maeda, "Improved Learning Performance of Hardware Self-Organizing Map Using a Novel Neighborhood Function," in IEEE Transactions on Neural Networks and Learning Systems, vol. 26, no. 11, pp. 2861-2873, Nov. 2015. doi: 10.1109/TNNLS.2015.2398932.

[2]Qing Y. Tang and Mohammed A. S. Khalid. 2016. Acceleration of k-Means Algorithm Using Altera SDK for OpenCL. ACM Trans. Reconfigurable Technol. Syst. 10, 1, Article 6 (September 2016), 19 pages. DOI: https://doi.org/10.1145/2964910.

[3]Momen, Mohammad Abdul. "FPGA-Based Acceleration of Expectation Maximization Algorithm using High Level Synthesis." MASc Thesis, University of Windsor, 2017.

[4]Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. 2018. A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks. J. Emerg. Technol. Comput. Syst. 14, 2, Article 18 (July 2018), 16 pages. DOI: https://doi.org/10.1145/3154839.

[5]Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. 2019. [DL] A Survey of FPGA-based Neural Network Inference Accelerators. ACM Trans. Reconfigurable Technol. Syst. 12, 1, Article 2 (March 2019), 26 pages. DOI: https://doi.org/10.1145/3289185.

[6]Ma, Yufei, Naveen Suda, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler." Integration 62 (2018): 14-23.

[7]Davidson, Gavin. "A parallel implementation of the self organising map using OpenCL." Level 4 Project, School of Computing Science, University of Glasgow (2015).

[8]Intel FPGA SDK for OpenCL, [Online] Available from - https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html. [Accessed August 13, 2019]

[9]Luthra, Siddhant. "High level synthesis and evaluation of an automotive radar signal processing algorithm for fpgas." MASc Thesis, University of Windsor, 2017.

[10]The Khronos Group Inc., [Online] Available from - https://www.khronos.org/. [Accessed August 13, 2019].

[11]Sanders, Jason, and Edward Kandrot. CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional, 2010.

[12]Altera SDK for OpenCL – Getting Started Guide. [Online] Available from - https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/ug-aocl-getting-started-16.0.pdf. [Accessed August 13, 2019].

[13]Altera SDK for OpenCL – Programming Guide.[Online] Available from - https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/aocl-programming-guide-15.1.pdf. [Accessed August 13, 2019].

[14]Altera SDK for OpenCL – Best Practices Guide. [Online] Available from - https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf. [Accessed August 13, 2019].

[15]What Is High-Performance Computing? [Online] Available from - https://www.netapp.com/us/info/what-is-high-performance-computing.aspx. [Accessed August 20, 2019]

[16]Geshi, M. (2019). The Art of High Performance Computing for Computational Science, Vol. 1. Springer.

[17]About High-Performance Computing (HPC). [Online] Available from - https://www.ichec.ie/news/press-corner/about-high-performance-computing-hpc. [Accessed August 20, 2019].

[18]Shan, Amar. "Heterogeneous processing: a strategy for augmenting moore's law." Linux Journal 2006, no. 142 (2006): 7.

[19]Zahran, M. (2019). Heterogeneous Computing: Hardware and Software Perspectives. Morgan & Claypool.

[20]Gottlieb, Allan, and G. Almasi. Highly parallel computing. Redwood City, CA: Benjamin/Cummings, 1989.

[21]Bit-level parallelism. [Online] Available from - https://en.wikipedia.org/wiki/Bit-level_parallelism. [Accessed August 20, 2019].

[22]Instruction-level parallelism [Online] Available from - https://en.wikipedia.org/wiki/Instruction-level_parallelism. [Accessed August 20, 2019].

[23]Data-level parallelism. [Online] Available from - https://en.wikipedia.org/wiki/Data_parallelism. [Accessed August 20, 2019].

[24]Task-level parallelism. [Online] Available from - https://en.wikipedia.org/wiki/Task_parallelism. [Accessed August 20, 2019].

[25]Inta, Ra, David J. Bowman, and Susan M. Scott. "The Chimera: an off-the-shelf CPU/GPGPU/FPGA hybrid computing platform." International Journal of Reconfigurable Computing 2012 (2012): 2.

[26]Waidyasooriya, H. M., Hariyama, M., & Uchiyama, K. (2018). Design of FPGA-based computing systems with OpenCL. Springer International Publishing.

[27]Parab, J. S., Gad, R. S., & Naik, G. M. (2018). Hands-on Experience with Altera FPGA Development Boards. Springer.

[28]FPGA Architecture. [Online] Available from - https://allaboutfpga.com/fpga-architecture/.[Accessed August 20, 2019].

[29]Stratix V FPGA Features. [Online] Available from - https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-v/features.html. [Accessed August 20, 2019].

[30]Arria 10 FPGA Features. [Online] Available from - https://www.intel.co.uk/content/www/uk/en/products/programmable/fpga/arria-10/features.html. [Accessed August 20, 2019].

[31]Terasic [Online] Available from - https://www.terasic.com.tw/en/.[Accessed August 20, 2019].

[32]Bittware [Online] Available from - https://www.bittware.com. [Accessed August 20, 2019].

[33]Altera Corporation, "OpenCL Reference Platforms," [Online]. [Accessed: Dec 01, 2015]

[34]Nallatech 385 –with Stratix V A7 FPGA PCIe Accelerator Card. [Online] Available from -

https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/isi-nallatech/board/385---with-stratix-v-a7-fpga-pcie-accelerator-card.html. [Accessed August 20, 2019]

[35]Nallatech 385A –with Arria 10 FPGA PCIe Accelerator Card. [Online] Available from - https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/isi-nallatech/board/385a---pga-accelerator-card-with-arria-10-fpga.html. [Accessed August 20, 2019]

[36]Coussy, Philippe, and Adam Morawiec, eds. High-level synthesis: from algorithm to digital circuit. Springer Science & Business Media, 2008.

[37]Nane, Razvan, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen et al. "A survey and evaluation of FPGA high-level synthesis tools." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35, no. 10 (2015): 1591-1604.

[38]Scarpino, M. (2011). OpenCL in action: how to accelerate graphics and computations. Manning Publications.

[39]The Khronos Group Inc. “OpenCL Overview” [online]. Available: https://www.khronos.org/opencl/. [Accessed August 20, 2019].

[40]A. Munshi, "The OpenCL specification," 2009 IEEE Hot Chips 21 Symposium (HCS), Stanford, CA, 2009, pp. 1-314.doi: 10.1109/HOTCHIPS.2009.7478342

[41]Janik, Ian, Qing Tang, and Mohammed Khalid. "An Overview of Altera SDK for OpenCL: A User Perspective." In 2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE), pp. 559-564. IEEE, 2015.

[42]Kohonen, Teuvo. "The self-organizing map." Proceedings of the IEEE 78, no. 9 (1990): 1464-1480.

[43]Kohonen, Teuvo. "Exploration of very large databases by self-organizing maps." In Proceedings of International Conference on Neural Networks (ICNN'97), vol. 1, pp. PL1-PL6. IEEE, 1997.

[44]Johnsson, Magnus, ed. Applications of Self-Organizing Maps. BoD–Books on Demand, 2012.

[45]Image of nervous system. [online] Available: https://pngtree.com/freepng/simple-human-nervous-system-map_3254163.html. [Accessed August 20, 2019]

[46]Image of human brain. [online] Available: https://en.wikipedia.org/wiki/Human_brain. [Accessed August 20, 2019].

[47]Roussinov, Dmitri G., and Hsinchun Chen. "A scalable self-organizing map algorithm for textual classification: A neural network approach to thesaurus generation." Communication Cognition and Artificial Intelligence, Spring (1998), v.15, pg.: 81-112.

[48]Richardson, A. J., Risien, C., & Shillington, F. A. (2003). Using self-organizing maps to identify patterns in satellite imagery. Progress in Oceanography, 59(2-3), 223-239.

[49]Hardman-Mountford, N. J., Richardson, A. J., Boyer, D. C., Kreiner, A., & Boyer, H. J. (2003). Relating sardine recruitment in the Northern Benguela to satellite-derived sea surface height using a neural network pattern recognition approach. Progress in Oceanography, 59(2-3), 241-255.

[50]Lobo, V. J. (2009). Application of self-organizing maps to the maritime environment. In Information Fusion and Geographic Information Systems (pp. 19-36). Springer, Berlin, Heidelberg.

[51]Iturriaga, F. J. L., & Sanz, I. P. (2013). Self-organizing maps as a tool to compare financial macroeconomic imbalances: The European, Spanish and German case. The Spanish Review of Financial Economics, 11(2), 69-84.

[52]Faigl, J. (2016). An application of self-organizing map for multirobot multigoal path planning with minmax objective. Computational intelligence and neuroscience, 2016.

[53]Marie Cottrell, Madalina Olteanu, Fabrice Rossi, Nathalie Villa-Vialaneix. Self-OrganizingMaps, theory and applications. Revista de Investigacion Operacional, 2018, 39 (1), pp.1-22. hal-01796059.

[54]McConnell, Sabine, Robert Sturgeon, Gregory Henry, Andrew Mayne, and Richard Hurley. "Scalability of Self-organizing Maps on a GPU cluster using OpenCL and CUDA." In Journal of Physics: Conference Series, vol. 341, no. 1, p. 012018. IOP Publishing, 2012.

[55]Moraes, Felipe C., Silvia C. Botelho, Nelson Duarte Filho, and Joel Felipe O. Gaya. "Parallel high dimensional self organizing maps using CUDA." In 2012 Brazilian Robotics Symposium and Latin American Robotics Symposium, pp. 302-306. IEEE, 2012.

[56]Prabhu, Raghavendra D. "SOMGPU: an unsupervised pattern classifier on graphical processing unit." In 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), pp. 1011-1018. IEEE, 2008.

[57]Zhongwen, Luo, Liu Hongzhi, Yang Zhengping, and Wu Xincai. "Self-organizing maps computing on graphic process unit." European Symposium on Artificial Neural Networks Bruges (Belgium), 2005.

[58]Hikawa, Hiroomi. "FPGA implementation of self organizing map with digital phase locked loops." Neural Networks 18, no. 5-6 (2005): 514-522.

[59]Brassai, S. T. "FPGA based hardware implementation of a self-organizing map." In IEEE 18th International Conference on Intelligent Engineering Systems INES 2014, pp. 101-104. IEEE, 2014.

[60] Appiah, K., Hunter, A., Meng, H., Yue, S., Hobden, M., Priestley, N., ... & Pettit, C. (2009, June). A binary self-organizing map and its FPGA implementation. In 2009 International Joint Conference on Neural Networks (pp. 164-171). IEEE.

[61]Intel FPGA SDK for OpenCL. [Online] Available from - https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html. [Accessed August 20, 2019]

[62]AMD Athlon II 170u Processor. [Online] Available from - https://www.cnet.com/products/amd-athlon-ii-x2-170u-2-ghz-processor/.[Accessed August 20, 2019]

[63]Nvidia Quadro K620 GPU. [Online] Available from - https://images.nvidia.com/content/pdf/quadro/datasheets/75509_DS_NV_Quadro_K620_US_NV_HR.pdf. [Accessed August 20, 2019]

[64]Gavin Davidson, Parallel Self-Organising Map in OpenCL, 2014-2015. [Online] Available from - https://github.com/wimvanderbauwhede/Parallel-SOM.

[65]Watts Up? Pro Power Meter. [Online] Available from - https://www.vernier.com/products/sensors/wu-pro/.[Accessed August 20, 2019]

Appendix A: Intel FPGA SDK for OpenCL Template

```
//Library and header file declaration
//Declaring all the necessary libraries and associated/required header
//files for the implementation of the program
#include <stdio.h>
#include <stdlib.h>
.
.
.
.
.
#include "CL/opencl.h"              //OpenCL header file
#include "AOCLUtils/aocl_utils.h"   //AOCL header file
#include "header_file.h"            //Header file created by the
                                    //user(if required)

using namespace std;
using namespace aocl_utils;         //AOCL declaration

//AOCL Alignment
//Required for DMA transfer from host to device [14]
#define AOCL_ALIGNMENT 64

//Enumeration of kernels
//Used as an identifier specifying the kernel number to be used in code
//efficiently.
enum KERNELS
{
    K_1,
    K_2,
    .
    .
    K_NUM_KERNELS
};

//Kernel names
//Lists the names of the kernels used in the kernel.cl while writing
//the kernel codes for each kernel.
static const char* kernel_names[K_NUM_KERNELS] =
{
    "kernel_1_name",
    "kernel_2_name",
    .,
    .,
    "last_kernel_name"
};

//Runtime OpenCL Configuration
//Used for declaring/creating platform, device, context, queues, kernel
//and program variables according to AOCL specifications
static cl_platform_id platform = NULL;          //Platform
static cl_device_id device = NULL;              //Device
static cl_context context = NULL;               //Context
static cl_command_queue queues[K_NUM_KERNELS];  //Queques
static cl_kernel kernels[K_NUM_KERNELS];        //Kernels
static cl_program program = NULL;               //Program
static cl_int status = 0;                       //Status for all OpenCL
                                                //execution

//Device Buffer
//Used to store data for FPGA implementation and transferring data to
and from host.
cl_mem d_buffer1;
cl_mem d_buffer2;
.
.
.
cl_mem d_budderN;


//Host Buffer
//Used to store data for CPU implementation and transferring data to
and from device.
cl_float * h_buffer1 = new cl_float;
cl_float * h_buffer2 = new cl_float;
.
.
.
cl_float * h_bufferN = new cl_float;

//Execution time variable
//Variables declared for performance computation.
float cpu_time = 0.0;           //CPU Execution time in s
double fpga_time = 0.0;         //FPGA Execution time in s
float start_time_cpu = 0.0;     //Start CPU execution at 0s
double start_time_fpga = 0.0;   //Start FPGA execution in 0s

//Function Prototype – Support
//Function generates the dataset for the program and conducts all
//initial calculations that is suitable to be implemented in CPU before
//FPGA implementation
void initialize();


//Function Prototype – CPU
//Function carries out the CPU version of the program. The output of
//this function will be compared with FPGA implementation for
//verification of results.
void run_cpu();         //CPU execution

//Function Prototype - OpenCL
bool init_opencl();     //Initialize device for OpenCL implementation
void run_fpga();        //FPGA execution
void cleanup();         //Release memory objects
```

```
// START: MAIN FUNCTION

int main()
{
    //Initializing OpenCL
    if(!init_opencl()) //Initializing OpenCL
    {
        return false;
    }

    printf("\nSUCCESSFUL: OpenCL FPGA Initialization.\n\n");

    //Initializing data
    initialize();

    //FPGA - Implementation
    run_fpga();

    //CPU - Implementation
    run_cpu();

    //Memory Cleanup
    cleanup();
}
//END: MAIN FUNCTION


//HELPER FUNCTIONS

// START: Initialize
void initialize()
{

    printf("\nSTART: Allocation of Host Buffer\n");

    //Allocating memory for host
    //The size of memory required for host buffer is declared here. The
    //host buffers needs to be 64-byte aligned in order to facilitate
    //DMA transfer to and from FPGA [14].
    int temp_h_buffer1 = posix_memalign((void**)&h_buffer1,
AOCL_ALIGNMENT, sizeof(cl_float));
    int temp_h_buffer2 = posix_memalign((void**)&h_buffer2,
AOCL_ALIGNMENT, sizeof(cl_float));
    .
    .
    .
    int temp_h_bufferN = posix_memalign((void**)&h_bufferN,
AOCL_ALIGNMENT, sizeof(cl_float));

    if(!temp_h_buffer1 || !temp_h_buffer2 || !temp_h_bufferN)
    {
        printf("\nSUCCESSFUL: Allocation of Host Buffer.\n");
    }
    else
    {
        printf("\nERROR: Allocation of Host Buffer.\n");
    }
    printf("Initialization SUCCESS!!\n");
}

// END: Initialize


//Initializing OpenCL

// START: OpenCL Initialization

bool init_opencl()
{
    cl_int status;

    //Start everything at NULL to help identify errors
    for(int i = 0; i < K_NUM_KERNELS; ++i)
    {
        kernels[i] = NULL;
        queues[i] = NULL;
    }

    //Locate Files via relative path
    if(!setCwdToExeDir())
    {
        return false;
    }

    //Get the OpenCL Platform
    //platform = findPlatform("Intel(R) FPGA");
    platform = findPlatform("Altera");
    if(platform == NULL)
    {
        printf("ERROR: Unable to find Intel(R) FPGA OpenCL
platform.\n");
        return false;
    }

    //Query the available OpenCL devices and just use the first device
if there is more than one
    scoped_array<cl_device_id> devices;
    cl_uint num_devices;
    devices.reset(getDevices(platform, CL_DEVICE_TYPE_ALL,
&num_devices));
    device = devices[0];

    //Create the context
    context = clCreateContext(NULL, 1, &device, &oclContextCallback,
NULL, &status);
    checkError(status, "ERROR: Failed to create context\n");

    //Create the command queues
    for(int i = 0; i < K_NUM_KERNELS; ++i)
    {
        queues[i] = clCreateCommandQueue(context, device,
CL_QUEUE_PROFILING_ENABLE, &status);
```

```
        checkError(status, "ERROR: Failed to create command queue (%d:
%s)\n", i, kernel_names[i]);
    }

    //Create the program
    std::string binary_file =
getBoardBinaryFile("Kernel_AOCX_file_name", device);
    printf("Using AOCX: %s\n\n", binary_file.c_str());
    program = createProgramFromBinary(context, binary_file.c_str(),
&device, 1);

    //Build the program that was just created.
    status = clBuildProgram(program, 0, NULL, "", NULL, NULL);
    checkError(status, "ERROR: Failed to build program.\n");

    //Create the kernel - name passed in here must match kernel name in
the original CL file, that was compiled into an AOCX file using the AOC
tool
    for(int i = 0; i < K_NUM_KERNELS; ++i)
    {
        kernels[i] = clCreateKernel(program, kernel_names[i], &status);
        checkError(status, "ERROR: Failed to create kernel (%d: %s)\n",
i, kernel_names[i]);
    }

    return true;
}
// END: OpenCL Initialization

// START: FPGA Implementation

void run_fpga()
{

    //Create Device Buffer
    //Allocates the memory size and indicates the memory bank to store
    //the data for FPGA implementation.
    d_buffer1 = clCreateBuffer(context, CL_MEM_READ_WRITE |
CL_MEM_BANK_1_ALTERA, sizeof(cl_float), NULL, &status);
    checkError(status, "ERROR: Failed to allocate input device buffer:
d_buffer1.!\n");
    d_buffer2 = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_BANK_2_ALTERA, sizeof(cl_float), NULL, &status);
    checkError(status, "ERROR: Failed to allocate input device buffer:
d_buffer2.!\n");
    .
    .
    .
    d_bufferN = clCreateBuffer(context, CL_MEM_READ_WRITE |
CL_MEM_BANK_1_ALTERA, sizeof(cl_float), NULL, &status);
    checkError(status, "ERROR: Failed to allocate input device buffer:
d_bufferN.!\n");

    printf("\nSUCCESSFUL: Created Device Buffer.!\n");
    //Copy Data from Host to Device
    //Transferring data from host buffers to device (i.e. FPGA) buffer
    //before starting the execution of the kernels.
    status = clEnqueueWriteBuffer(queues[Kernel_number], d_buffer1,
CL_TRUE, 0, sizeof(cl_float), h_buffer1, 0, NULL, NULL);
    checkError(status, "ERROR: Failed to copy data from host to device:
h_buffer1, d_buffer1.!\n");
    .
    .
    .
    status = clEnqueueWriteBuffer(queues[Kernel_number], d_bufferN,
CL_TRUE, 0, sizeof(cl_float), h_bufferN, 0, NULL, NULL);
    checkError(status, "ERROR: Failed to copy data from host to device:
h_bufferN, d_bufferN.!\n");

    //Set the kernel argument
    //Sets the arguments of the kernels written in the kernel.cl file.
    status = clSetKernelArg(kernels[Kernel_number], <Argument_number>,
sizeof(cl_mem), (void*)&d_buffer1);
    checkError(status, "\nERROR: Failed to set up kernel (K_comp)
argument <Argument_number>\n");
    .
    .
    .
    status = clSetKernelArg(kernels[Kernel_number], <Argument_number>,
sizeof(cl_mem), (void*)&d_bufferN);
    checkError(status, "\nERROR: Failed to set up kernel (K_comp)
argument <Argument_number>\n");


    //Launching Kernel
    //Execution of the kernels starts from here on the FPGA board.
    status = clEnqueueTask(queues[Kernel_number],
kernels[Kernel_number], 0, NULL, NULL);
    checkError(status, "ERROR: Failed to launch kernel: %s\n",
kernel_names[Kernel_number]);
    //or
    status = clEnqueueNDRangeKernel(queues[Kernel_number],
kernels[Kernel_number], 1, NULL, <global_size>, <local_size>, 0, NULL,
NULL);
    checkError(status, "ERROR: Failed to launch kernel: %s",
kernel_names[Kernel_number]);


    //Finishing Command Queue of kernel
    //Waits for the execution of kernel to finish and then procedes to
    //the next step
    status = clFinish(queues[Kernel_number]);
    checkError(status, "\nERROR: Failed to finish command queue of
(%s)\n", kernel_names[Kernel_number]);


    //Reading from Device to Host
    //After the execution of the kernel is finished. Sends data from
    //the FPGA back to the CPU for analysis.
    status = clEnqueueReadBuffer(queues[Kernel_number], d_bufferN,
CL_TRUE, 0, sizeof(cl_float), h_bufferN, 0, NULL, NULL);
```

```
    checkError(status, "\nERROR: Failed to copy data from Device to
Host\n");

    status = clFinish(queues[Kernel_number]);
    checkError(status, "\nERROR: Failed to finish command queue of
(%s)\n", kernel_names[Kernel_number]);

}
// END: FPGA Implementation


//CLEANUP - Release Memory Objects

// START: Cleanup
//Releases the memory objects.
void cleanup()
{

    //Release kernels
    for(int i = 0; i < K_NUM_KERNELS; ++i)
    {
        if(kernels[i])
        {
            clReleaseKernel(kernels[i]);
        }
    }

    //Release Program
    if(program)
    {
        clReleaseProgram(program);
    }

    //Release command queue
    for(int i = 0; i < K_NUM_KERNELS; ++i)
    {
        if(queues[i])
        {
            clReleaseCommandQueue(queues[i]);
        }
    }

    //Release context
    if(context)
    {
        clReleaseContext(context);
    }

    //Free/release device buffer
    if(d_buffer1)
    {
        clReleaseMemObject(d_buffer1);
    }
    if(d_bufferN)
    {
        clReleaseMemObject(d_bufferN);
    }
    .
    .
    .
    if(d_bufferN)
    {
        clReleaseMemObject(d_bufferN);
    }
}
```

// END: Cleanup

Appendix B: SOM OpenCL Kernel

```
//inclusing header
#include "../host/inc/som.h"

//*****************************************************************
//START >> Kernel 1
//*****************************************************************
__kernel
__attribute__((task))
void SOMComp(           __global float * restrict K_cur_map,
                        __global float * restrict K_input,
                        __global float * restrict K_gauss_value_list)
{

    int input_index;
    int winnerpass = 0;
    int winner = 0;
    float winnerDistance;
    float possible_winnerDistance;
    int current_pos;
    int neighbourhood_value;
    int a_x;
    int a_y;
    int b_x;
    int b_y;
    int output;
    int total_map_values_fpga =
map_side_size*map_side_size*input_vector_length;
    int total_input_values = input_size*input_vector_length;
    float g_gauss[map_side_size];
    __local float g_distance_map[map_side_size*map_side_size];
    __local float
cur_map[map_side_size*map_side_size*input_vector_length];
    __local float g_input[input_size*input_vector_length];

    #pragma unroll map_side_size
    for(int j = 0; j < total_input_values; j++)
    {
        g_input[j] = K_input[j];
    }

    #pragma unroll map_side_size
    for(int i = 0; i < total_map_values_fpga; i++)
    {
        cur_map[i] = K_cur_map[i];
    }

    #pragma unroll map_side_size
    for(int i = 0; i < map_side_size; i++)
    {
        g_gauss[i] = K_gauss_value_list[i];
    }
    for(input_index = 0; input_index < total_input_values; input_index
= input_index + input_vector_length)
    {
        float sum = 0;
        int b_index = input_index;
        #pragma unroll input_vector_length
        for(int a_index = 0; a_index < input_vector_length; a_index++)
        {
            sum += fabs(cur_map[a_index] - g_input[b_index]);
            b_index++;
        }
        winnerDistance = sum;


        for(int i = 0; i < total_map_values_fpga; i = i +
input_vector_length)
            {
                float accu = 0;
                int c_index = input_index;

                for(int j = i; j < (input_vector_length + i); j++)
                {
                    accu += fabs(cur_map[j] - g_input[c_index]);
                    c_index++;
                }
                g_distance_map[i/input_vector_length] = accu;
            }
        #pragma unroll map_side_size
        for(int distance_index = 0; distance_index <
(map_side_size*map_side_size); distance_index++)
        {
            if(g_distance_map[distance_index] < winnerDistance)
            {
                winnerDistance = g_distance_map[distance_index];
                winner = distance_index;
            }
        }
        winnerpass = winner;


        for(int i = 0; i < total_map_values_fpga; i++)
        {
            int a = i/input_vector_length;
            int b = winnerpass;

            a_x = a % map_side_size;
            a_y = a / map_side_size;
            b_x = b % map_side_size;
            b_y = b / map_side_size;

            neighbourhood_value = max(abs(a_x - b_x), abs(a_y - b_y));

            cur_map[i] = cur_map[i] - ((cur_map[i] -
g_input[input_index + (i % input_vector_length)]) *
g_gauss[neighbourhood_value]);
        }
    }
    #pragma unroll map_side_size
    for(int l = 0; l < total_map_values_fpga; l++)
    {
        K_cur_map[l] = cur_map[l];
    }
}

//**********************************************************************
//END >> Kernel 1
//**********************************************************************

//**********************************************************************
//START >> Kernel 2
//**********************************************************************

__kernel
__attribute__((task))
void NeigRed(          __global float * restrict K_gauss_value_list)
{

    float temp_value;
    float g_gauss[map_side_size];


    #pragma unroll map_side_size
    for(int j = 0; j < map_side_size; j++)
    {
        g_gauss[j] = K_gauss_value_list[j];
    }


    #pragma unroll map_side_size
    for(int i = 1; i < map_side_size; i++)
    {
        temp_value = g_gauss[i];
        g_gauss[i - 1] = temp_value;
    }
    g_gauss[map_side_size - 1] = 0;

    #pragma unroll map_side_size
    for(int k = 0; k < map_side_size; k++)
    {
        K_gauss_value_list[k] = g_gauss[k];
    }
}

//**********************************************************************
//END >> Kernel 2
//**********************************************************************
```

Vita Auctoris

NAME:	Mohammad Abdul Moin Oninda
PLACE OF BIRTH:	Dhaka, Bangladesh
YEAR OF BIRTH:	1995
EDUCATION:	Bachelor of Science in Electrical & Electronic Engineering (2014-2017) Islamic University of Technology, Gazipur, Bangladesh.
	Master of Applied Science in Electrical Engineering (2019) Department of Electrical and Computer Engineering, University of Windsor, Windsor, ON, Canada.