VISVESVARAYA TECHNOLOGICAL UNIVERSITY

BELAGAVI - 590018

Project Report

on

"CUSTOM ACCELERATOR FOR GRAPH ANALYTICS"

Submitted in partial fulfilment of the requirements for the VIII Semester

Bachelor of Engineering

in

ELECTRONICS AND COMMUNICATION ENGINEERING

For the Academic Year

2020-2021

BY

ANNETTE ANTONY	**1PE17EC018**
AVANI S	**1PE17EC027**
BHIMALA SUBBARAYUDU	**1PE17EC031**
JEEVAN R	**1PE17EC055**

UNDER THE GUIDANCE OF

DR. MADHURA PURNAPRAJNA

PROFESSOR, Dept. of ECE, PESIT-BSC

Department of Electronics and Communication Engineering

PESIT - BANGALORE SOUTH CAMPUS

Hosur Road, Bengaluru - 560100

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

BELAGAVI - 590018

Project Report

on

"CUSTOM ACCELERATOR FOR GRAPH ANALYTICS"

Submitted in partial fulfilment of the requirements for the VIII Semester

Bachelor of Engineering

in

ELECTRONICS AND COMMUNICATION ENGINEERING

For the Academic Year

2020-2021

BY

ANNETTE ANTONY	**1PE17EC018**
AVANI S	**1PE17EC027**
BHIMALA SUBBARAYUDU	**1PE17EC031**
JEEVAN R	**1PE17EC055**

UNDER THE GUIDANCE OF

DR. MADHURA PURNAPRAJNA

PROFESSOR, Dept. of ECE, PESIT-BSC

Department of Electronics and Communication Engineering

PESIT - BANGALORE SOUTH CAMPUS

Hosur Road, Bengaluru - 560100

PESIT - BANGALORE SOUTH CAMPUS

HOSUR ROAD, BENGALURU - 560100

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

CERTIFICATE

This is to certify that the project work entitled **"Custom Accelerator for Graph Analytics"** carried out by **Annette Antony, Avani S, Bhimala Subbarayudu, Jeevan R** bearing USNs **1PE17EC018, 1PE17EC027, 1PE17EC031, 1PE17EC055** respectively in partial fulfillment for the award of Degree of Bachelors (Bachelor of Engineering) in Electronics and Communication Engineering of Visvesvaraya Technological University, Belagavi during the year 2020-2021. It is certified that all corrections/ suggestions indicated for internal assessment have been incorporated in the report. The project report has been approved as it satisfies the academic requirements in respect of project work prescribed for the said Degree.

Signature of the Guide	Signature of the HOD	Signature of the Principal
Dr. Madhura Purnaprajna	**Dr. Subhash Kulkarni**	**Dr. Subhash Kulkarni**
Professor	**HOD, ECE**	**Principal, PESIT-BSC**

External Viva

Name of the Examiners	**Signature with Date**
1.	
2.	

Acknowledgements

We would like to express our sincere gratitude to the faculty and staff of the Department of Electronics and Communication Engineering for providing us with their help and guidance.

We would like to thank the college management and **Dr. Subhash Kulkarni**, Principal of PESIT-BSC and Head of Department of Electronics and Communication Engineering for encouraging and giving us the opportunity to work on this project.

We would like to convey our special thanks of gratitude to our guide **Dr. Madhura Purnaprajna** for her assistance, encouragement and her constant support which made this project possible.

We are also thankful to our project coordinators, **Prof. Lokesh L** and **Prof. Rajesh C**, for providing us with all needed information and facilities whenever it was requested.

Lastly, we would like to thank our peers and families, who made working on this project easier.

ABSTRACT

In recent years, the size of graph data sets has been increasing exponentially and this massive data scaling is causing a driving need for system designers and developers to invent computationally efficient systems with low power consumption, effective memory management schemes, and are economical. Most of the real-world problems can be modeled as a graph having millions, in some cases even trillions, of vertices and edges. Typical examples of using graphs are in the fields of machine learning, social networks, medical sciences, pharmaceuticals, and many others.

Irregular memory access patterns of graph algorithms, poor memory access locality, irregular and unstructured nature of graph problems make it difficult to take advantage of fast on-chip memory, such as cache memory in CPUs. By designing a tailored FPGA framework for given graph algorithms, we can mitigate some of the above inconveniences to swiftly process graph data to save execution time and power consumed, thereby achieving high performance and significantly improved throughput.

In this project, we have considered a few popular graph algorithms– PageRank, Single Source Shortest Path (SSSP), Breadth First Search (BFS), and Depth First Search (DFS). We had employed the Vivado HLS tool and its optimization methodologies to design an FPGA accelerator for the respective algorithms and verified the functionality on the hardware using the PYNQ-Z2 FPGA Development Board. Due to constraints on the on-chip resources of the device, we have adopted algorithm-specific graph partitioning schemes to process large graphs. We have used the GAP Benchmark Suite running on CPU hardware as the baseline for evaluating the performance of our design. Simulation results have shown significant improvement in the latencies between the CPU benchmark vs. our FPGA accelerator.

Keywords: Graphs, Graph algorithms, Hardware Accelerator, Graph Analytics, Resource-constrained devices, Graph Partitioning, PageRank, SSSP, BFS, DFS, GAP Benchmark

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What is VLSI?

Very large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining millions of transistors onto a single chip, thus permitting complex semiconductor technologies to be designed and developed.
Some of the examples of VLSI devices are microprocessors, microcontrollers, memory chips, Field-programmable gate arrays, or just FPGAs.

1.2 What is a Graph?

Graph is a data structure consisting of a set of vertices/nodes (V) and edges (E). Vertices model objects and edges model relationships between objects.
Mathematically, graph G is represented as an ordered pair of its vertices and edges : G = (V, E), where 'V' is a finite set of vertices and 'E' is a finite set of edges.

In the example shown in Figure 1.1, a graph with 5 vertices and 6 edges is illustrated. This graph G can be defined as G = (V , E),
where V = {A, B, C, D, E}
and E = {(A,B), (A,C), (A,D), (B,D), (C,D), (B,E) ,(E,D)}.

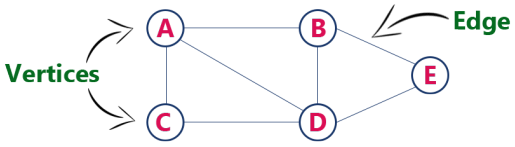


Figure 1.1: A simple graph data structure

1.2.1 Types of Graphs

Graphs are broadly classified into four intermingling types. They are:

1. **Undirected graph** : All the edges connecting nodes are implicitly bidirectional. It means that in an undirected graph, if there is an edge from vertex 'A' to vertex 'B', it is implied that there is an edge connecting vertex 'B' to 'A' also.

2. **Directed graph** : All the edges connecting nodes are unidirectional. We can establish bidirectionality between two nodes by connecting them explicitly by two edges in opposite directions.

3. **Weighted graph** : Each edge in the graph is assigned a real numerical value called "weight".

4. **Unweighted graph** : All the edges in the graph have no "weight" attached to them. Or we can also say that the weight of each edge in an unweighted graph is unity.

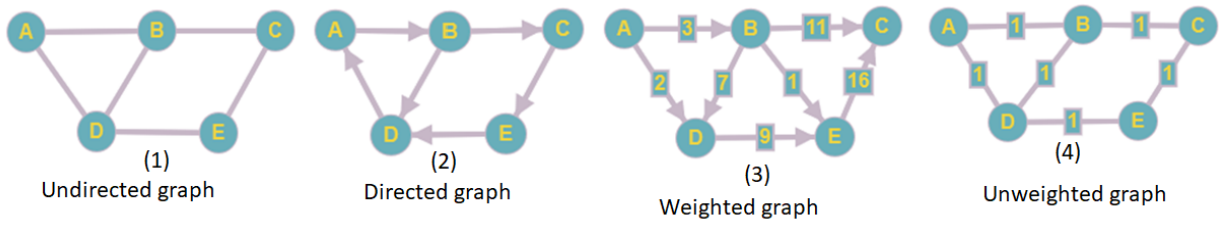


Figure 1.2: Types Of Graphs

1.2.2 Graph Representations

There are three widely used formats for representing the graph data sets. They are:

1. **Adjacency Matrix**: It is a two dimensional array, A, (or a square matrix) of size V x V.
   A[i][j] = 1 if edge exists between nodes 'i' and 'j' and A[i][j] = 0 if no connecting edge exists between them.
   Row indices correspond to the vertices and column indices correspond to their neighbors.

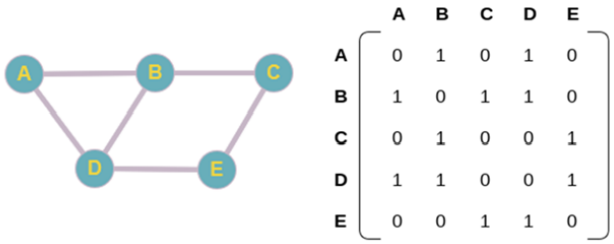


Figure 1.3: Adjacency Matrix

2. **Adjacency List** : It is a **'linked list'** representation of a graph. The size of the array is equal to the number of vertices in the graph. Each element in the array is the a pointer to a linked list made up of its unordered neighboring nodes.

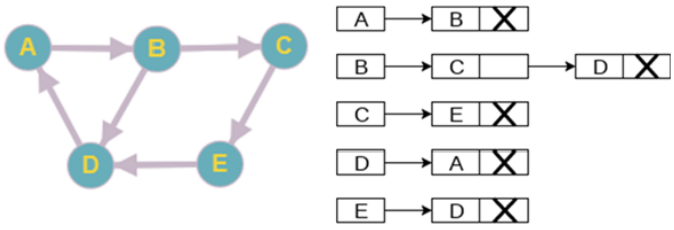


Figure 1.4: Adjacency List

3. **Edge List** : In this format, a graph is represented as a list of edges. Each element of the list has two entries for an unweighted graph, where first value is the source vertex and the second value is the destination vertex, and three entries for a weighted graph (where the third value represents the weight of the edge).

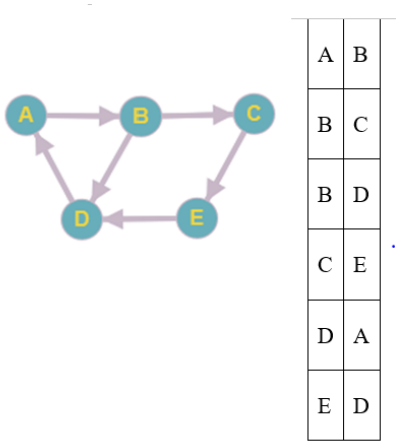


Figure 1.5: Edge List

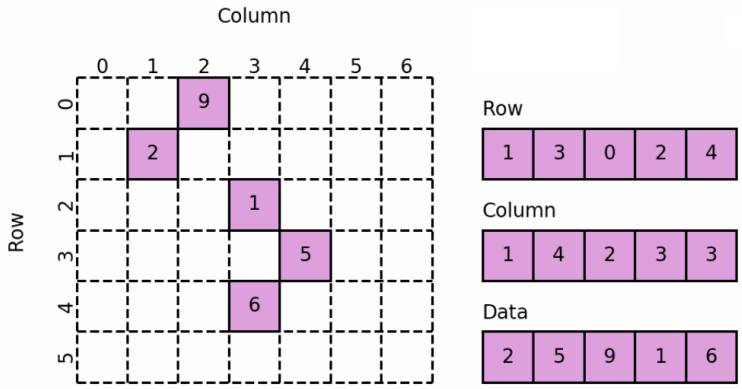
1.2.3 Sparse Matrix Representations

Most of the real world graphs have sparse adjacency matrices. The sparsity is defined as the proportion of zeros in the adjacency matrix. The sparsity of the real world graph data sets with millions of nodes and edges is between 97% to 99%.

Some of the graph algorithms like PageRank that work on adjacency matrix representation of graphs suffer in performance from this inherent sparsity because of redundant calculations. To overcome this redundancy, dense representations of sparse matrices are used wherein we represent the adjacency matrix with only its non-zero elements.

There are three such formats that are commonly used. They are:

1. **Co-Ordinate (COO) format** : It consists of three 1D arrays, namely, the 'Data' array which stores the non-zero values of the adjacency matrix and the corresponding row and column indices are stored in the 'Row' and 'Column' array respectively.

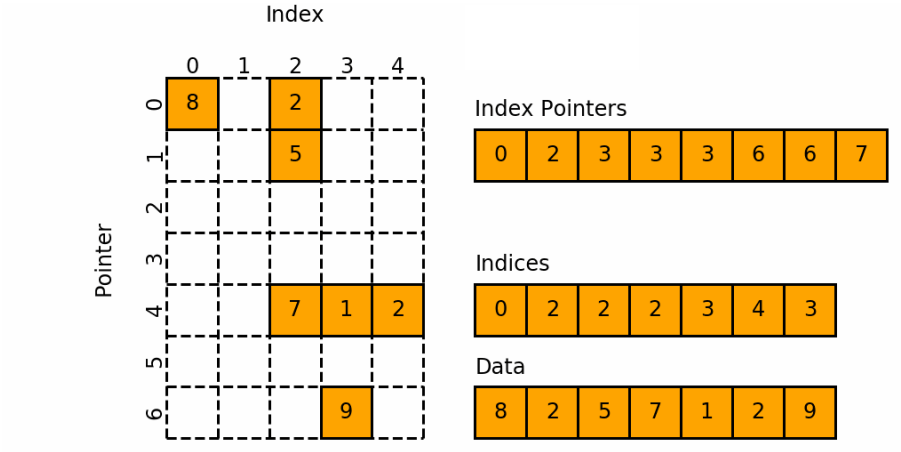


Source : Google Images

Figure 1.6: Co-Ordinate format

2. **Compressed Sparse Row (CSR) format** : It is similar to COO, but compresses the row indices, hence the name. This format allows fast row access and matrix-vector multiplications.
   The CSR format stores a sparse matrix **m x n** in row form using three 1D arrays (Data, Indices, Index_Pointers). Let NNZ denote the number of nonzero elements in the matrix. The arrays 'Data' and 'Indices' are of length NNZ, and contain the non-zero values and the column indices of those values respectively.

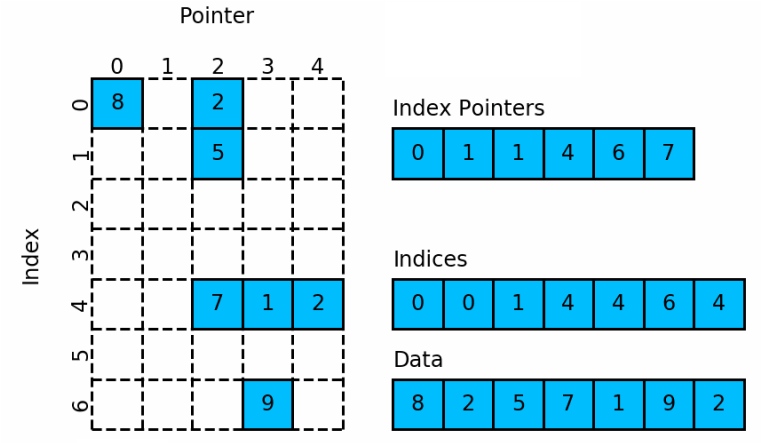
The array 'Index_Pointers' is of length m + 1 and encodes the index in 'Data' and 'Indices' where the given row starts. The last element is NNZ.



Source : Google Images

Figure 1.7: Compressed Sparse Row format

3. **Compressed sparse Column(CSR) format** : CSC works exactly the same as CSR but has column based index pointers and row indices instead. In the example illustrated in Figure 1.8, 'Data' is an array of the (top-to-bottom, then left-to-right) non-zero values of the matrix; 'Indices' is the row indices corresponding to the values; and 'Index_Pointers' is the list of 'Data' indices where each column starts.
   The name is based on the fact that column index information is compressed relative to the COO format. This format is efficient for arithmetic operations, column slicing, and matrix-vector products.



Source : Google Images

Figure 1.8: Compressed Sparse Column format

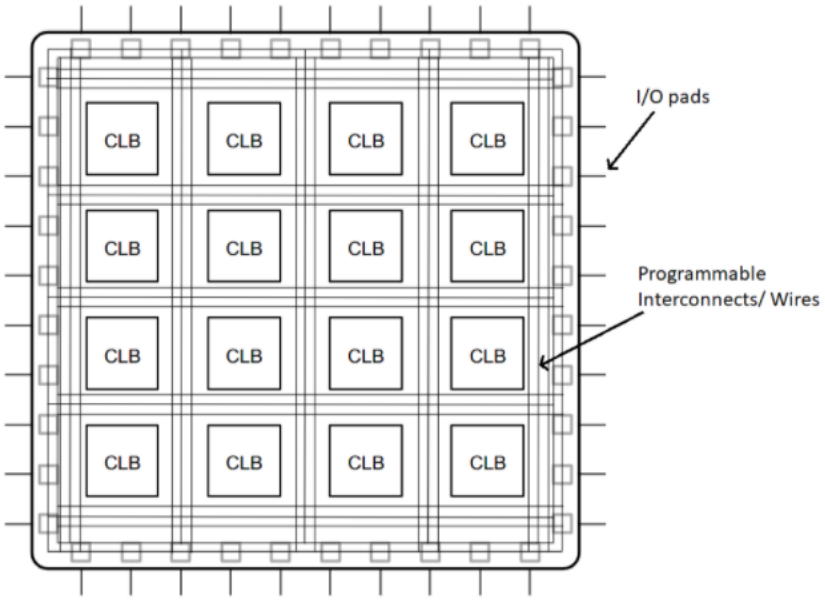
	Adjacency Matrix	Edge List	Adjacency List	CSR
Storage Cost	$O(n^2)$	$O(m)$	$O(m+n)$	$O(m+n)$
Add an Edge	$O(1)$	$O(1)$	$O(deg(v))$	$O(m+n)$
Delete an Edge from v to w	$O(1)$	$O(m)$	$O(deg(v))$	$O(m+n)$
Finding neighbours of v	$O(n)$	$O(m)$	$O(deg(v))$	$O(deg(v))$
Finding if w is neighbour of v	$O(1)$	$O(m)$	$O(deg(v))$	$O(deg(v))$

- m : No. of Edges
- n : No. of Nodes
- deg(v) : degree or the number of neighbors of vertex 'v'

Table 1.1: Tradeoffs in Some Graph Representations

1.3 What is an FPGA?

Field Programmable Gate Arrays (FPGAs) are a type of integrated circuits (IC) that can be reprogrammed for different algorithms and applications after fabrication. They are based around a matrix of configurable logic blocks (CLBs) – made up of LUTs and FFs – connected via programmable interconnects.



Source: Xilinx Documentation - Introduction to FPGA Design with Vivado High-Level Synthesis

Figure 1.9: Basic FPGA Architecture

The basic structure of an FPGA is composed of the following elements:

1. **Look-up table (LUT)** : This is the element that is responsible to perform

logic operations.

2. **Flip-Flop (FF)** : This register element is used to store the result of the LUT.
3. **Wires** : These elements connect other elements in the FPGA to one another.
4. **Input/Output (I/O) pads** : These are physical ports that are configured to get data in and out of the FPGA.

1.3.1 FPGA Design Flow

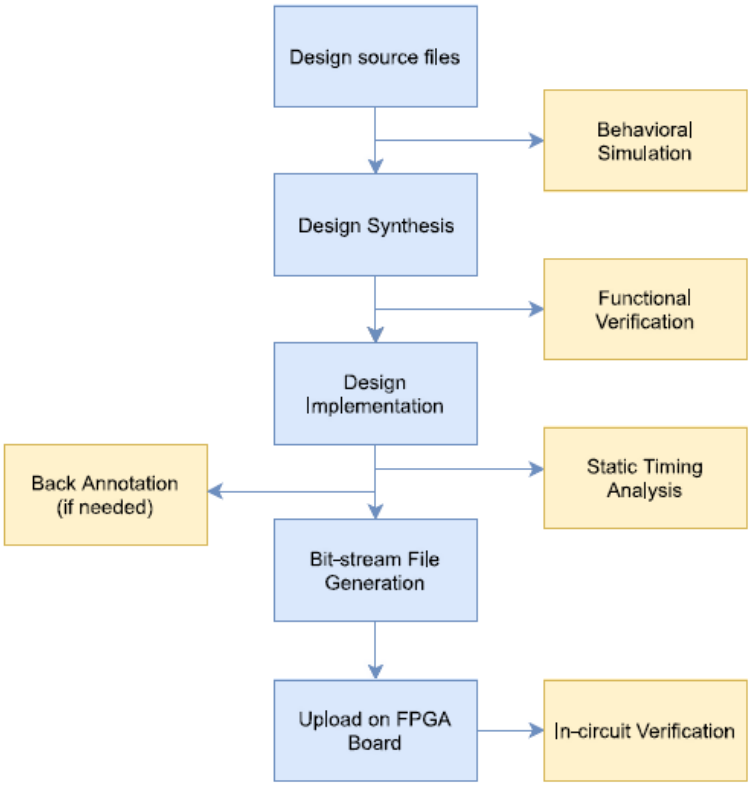


Figure 1.10: FPGA Design Flow

1. **Design Entry** : Design source files that describe the abstract functionality are written in Hardware Description Language (HDL). Design Constraints are written in an XDC file.
2. **Behavioral Simulation** : The behavior of the system is verified by running the behavioral simulation by means of a test bench.
3. **Design Synthesis** : The synthesis tool transforms the RTL code into a gate-level netlist.

4. **Design Implementation** : The implementation phase is a 3-step process. The translate stage does port assignments to actual FPGA pins, switches and buttons. The map phase divides the netlist into sub-blocks and maps them to actual FPGA blocks. Place and Route stage places the design sub-blocks in their designated FPGA blocks, then routes between the blocks.
5. **Static Timing Analysis** : STA is carried out after the design implementation to check that the design follows the timing constraints.
6. **Bit-stream File Generation** : The implemented design must then be converted into a Bitstream so that the FPGA platform can understand the design. It is uploaded to the board through the hardware manager.

Chapter 2

Literature Survey

Table 2.1: Literature Survey

Sl No	Resource Name and Authors	Content Applied
Research Papers		
1	**Graph Processing on FPGAs: Taxonomy, Survey, Challenges** by Maciej Besta, Dimitri Standojevic *et al*	Sufficient background in Graphs, its representations, taxonomy and FPGAs; challenges in graph processing; and extensive survey about related work done on efficient solutions to certain Graph Problems using FPGAs is presented.
2	**A reduced-precision streaming SpMV architecture for Personalized PageRank on FPGA** by Alberto Parravicini, Francesco Sgherzi and Marco D. Santambrogio.	Implements Personalized PageRank, on an FPGA by leveraging dataflow computation and reduced precision fixed-point arithmetic. It uses COO format and Sparse Matrix-Vector Multiplication.
Continued on next page		

Table 2.1 – continued from previous page

Sl No	Resource Name and Authors	Content Applied
3	**A Study of Partitioning Policies for Graph Analytics on Largescale Distributed Platforms** by Gurbinder Gill, Roshan Dathathri, Loc Hoang and Keshav Pingali	Introduction to graph partitioning strategies such as Edge-Cuts and Cartesian Vertex Cuts. Reduces Communication overhead by using proxies and also develops a decison tree to help choose a graph partitioning scheme.
4	**Accelerating PageRank using Partition-Centric Processing** by Kartik Lakhotia, Rajgopal Kannan and Viktor Prasanna.	Demonstrates Partition centric processing with Gather-Apply-Scatter model to improve the bandwidth utilization and reduce unwanted DRAM accesses to improve the performance of the PageRank algorithm.
5	**Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite** by Ariful Azad, Mohsen Mahmoudi Aznavehy, Scott Beamer *et al*.	Explains the GAP BenchMark Suite (containing PageRank), which is used by me to compare my code's performance against an existing benchmark. It consists of 30 tests: 6 graph algorithms on 5 graphs.
6	**An FPGA Framework for Edge-Centric Graph Processing** by Shijie Zhou, Rajgopal Kannan, Hanqing Zeng and Viktor K. Prasanna.	Designed an FPGA framework to determine the optimal design parameters and produce an optimized RTL FPGA accelerator design. It also optimises the data layout by exploiting thread level parallelism.

Continued on next page

Table 2.1 – continued from previous page

Sl No	Resource Name and Authors	Content Applied
7	**Optimizing Memory Performance for FPGA Implementation of PageRank** by Shijie Zhou, Charalampos Chelmis and Viktor K. Prasanna.	Introduces data layout optimisation and reduces pipeline stalls by dividing the input graph into vertex set, edge set and update set. It improves the performance of the algorithm by 70%.
8	**The PageRank Citation Ranking: Bringing Order to the Web** by Lawrence Page, Sergey Brin, Rajeev Motwani and Terry Winograd.	The very first paper introducing and defining the PageRank algorithm and the mathematical modelling behind it. It deploys the role of a random internet surfer and structures the web accordingly.
9	**ThunderGP: HLS-based Graph Processing Framework on FPGAs** by Xinyu Chen1, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong and Deming Chen.	Introduces an open-source HLS-based graph processing framework on FPGAs which adopts the Gather-Apply- Scatter (GAS) model. It evaluates 7 common graph algorithms and delivers 2.9x speedup compared to the state-of-the-art model.
Books		
1	**Parallel Programming for FPGAs** by Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer.	Explains the various optimisations available in HLS and demonstrates it's impact by applying them on various algorithms throughout the book.
Manuals		

Continued on next page

Table 2.1 – continued from previous page

Sl No	Resource Name and Authors	Content Applied
1	**Vitis High-Level Synthesis User Guide.** by Xilinx	Reference Guide to the Vivado HLS software by explaining the basics and the libraries available. It also demonstrates the coding styles to help improve the performance of a code.
2	**Vitis Unified Software Platform Documentation.** by Xilinx	Introduction to the Vitis Unified Software Platform and includes the methodology to accelerate applications using the Vitis Software Platform
3	**Vivado Design Suite User Guide.** by Xilinx	Introduction to Vitis HLS and provides a guide to migrate from Vivado HLS to Vitis HLS. Contains a list of coding styles and optimisations techniques available.
Presentations		
1	**Vivado HLS – Tips and Tricks** by Frédéric Rivoallon.	Explains how each optimisations work in detail and the importance of FPGAs in computing.

2.1 Problem Statement

With the ever increasing size of graph data sets rises the need for computationally efficient graph algorithms which consume less power and exploit the sparse nature of these data sets. Therefore, the problem statement for our project is to design an FPGA accelerator for graph processing which improves the performance of its CPU counterpart.

2.2 Solution

The solution to the aforementioned problem statement is to develop a custom FPGA framework for each of the four graph algorithms where we will be using a High Level Synthesis (HLS) tool to do fast prototyping and shorten the design cycle for FPGAs. Also, we are required to come up with graph partitioning schemes suitable for the given graph algorithm to process large data sets.

2.3 Objective

The main objective of our project is to demonstrate that by designing custom-made FPGA accelerators for graph processing we can alleviate most of the difficulties faced by CPUs and GPUs due to the inherent nature of graph problems and data sets.

Chapter 3

Hardware and Software Requirements and Specifications

3.1 Hardware Specifications

PYNQ-Z2 Development Board
Default part : xc7z020clg400-1 Product : Zynq-700 Family : Zynq-7000 Package : clg400 Speed grade : -1

TUL PYNQ™-Z2 board, based on Xilinx Zynq SoC, is designed for the Xilinx University Program to support PYNQ (Python Productivity for Zynq) framework and embedded systems development.

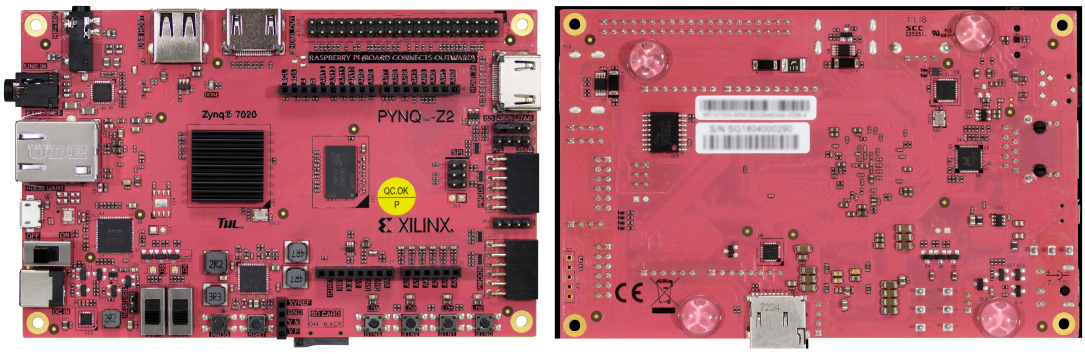


Figure 3.1: PYNQ-Z2 Board

The specifications of the board are as follows:
It has a dual-core Cortex-A9 processor running at a maximum frequency of 650MHz, with 13,300 logic slices, each with four 6-input LUTs and 8 flip-flops, 630 KB of fast block RAM, 220 DSP slices, and 512MB DDR3 memory. It has a DDR3 memory controller with 8 DMA channels and 4 High Performance AXI3

Slave ports. It can be powered from USB or 7V-15V external power source. It has 4 push-buttons, 2 slide switches, 4 LEDs, and 2 RGB LEDs among many others.

3.2 Software Specifications

3.2.1 Vivado HLS 2019.1 WebPack

The Xilinx Vivado HLS tool synthesizes a C function into an IP block that can be integrated into a hardware system.

Following is the Vivado HLS design flow:

1. Open Vivado HLS and create a new Vivado HLS project. Add the source code and testbench files and verify the functionality of the source code using the testbench with C simulation.
2. Run high-level synthesis on the source code to generate RTL files. Analyze the results of the C synthesis by examining latency, initiation interval (II), throughput, and resource utilization.
3. Optimize using HLS directives and repeat steps 2, 3, and 4 as needed and verify the results of the source code using C/RTL Co-simulation.
4. Package the RTL implementation of the application into an IP format for future implementation on the hardware.

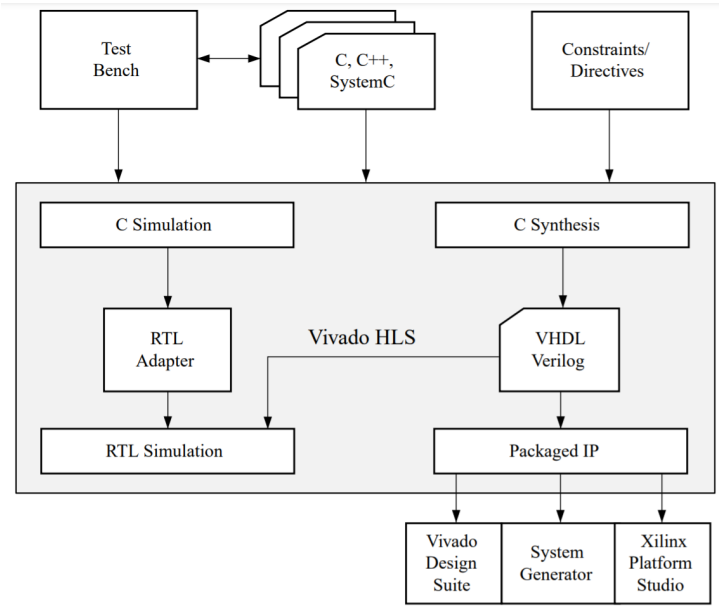


Figure 3.2: Vivado HLS Design Flow

Input to Vivado HLS :

1. Kernel code written in C, C++, SystemC .
2. Constraints include the clock period and target FPGA.
3. Directives are optional and direct the synthesis process to implement a specific behavior or optimization.
4. C test bench for verification of RTL implementation.

Output of Vivado HLS :

1. Primary output of Vivado HLS is the RTL implementation files in an hardware description language (HDL). The tool produces RTL implementations in both Verilog and VHDL.
2. Reports that contain the results of synthesis, C/RTL co-simulation, and IP packaging.

3.2.2 Vivado Design Suite

We will be exporting the IP package from the Vivado HLS with RTL implementation of our C algorithm to Vivado Design Suite for design Implementation, bitstream file generation, and Uploading onto the FPGA Board. Below are the steps for integrating your exported IP and generating bitstream for validating your design on the hardware.

1. Open Vivado Design Suite and click on ‘Create Project’ on the homepage. A dialog box will appear that will guide you through the creation of a new project.
2. Enter any name for your project and an easy-to-navigate location for your project and click ‘Next’. In the ‘Project Type’, choose ‘RTL Project’ and check the box under it that says ‘Do not specify sources at this time’ and click ‘Next’.
3. In ‘Default Part’, select the Board/Part that you created your HLS design for and click ‘Next’, then ‘Finish’. Your project will be created and it opens with a ‘Project Manager’ pane and a ‘Flow Navigator’ pane.

4. Click on ‘Create Block Design’ under ‘IP INTEGRATOR’ in the Flow Navigator pane and click ‘OK’.

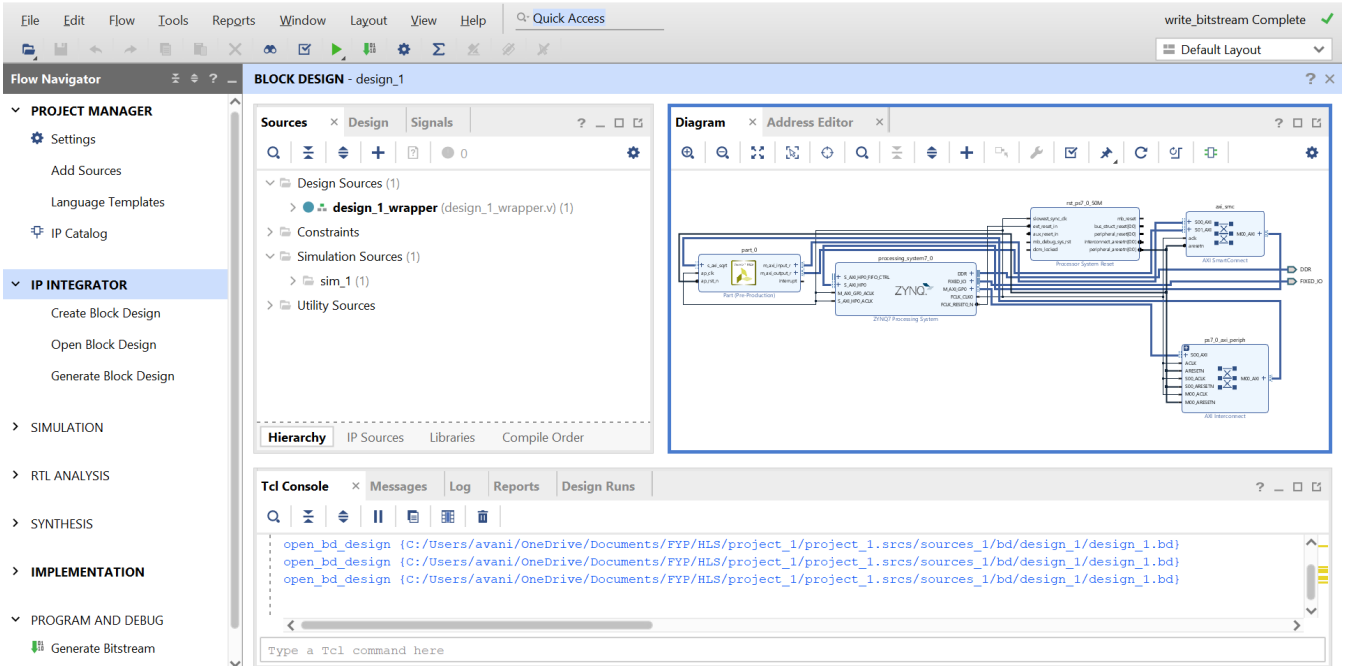


Figure 3.3: Project Window in Vivado Design Suite 2019.1

5. Add **‘ZYNQ7 Processing System’** IP to in the ‘Diagram’ window. Click on ‘Run Block Automation’ and click on ‘OK’ in the dialog box that appears.
6. Go to settings and navigate to IP Repository. Add the exported IP that you created in HLS and click on ‘Apply’ and then ‘OK’.
7. Add the exported IP to the Block Design and click on ‘Run Connection Automation’.
8. Double click on the Zynq7 PS and turn on the necessary interfaces needed according to your design interfaces. Run the connection automation again and validate your block design by clicking on F9 key on your keyboard.
9. Resolve any errors or critical warnings that appear while validating your design.
10. Navigate to the Sources and right click on the design file. Click on **‘Create HDL Wrapper’** and select auto-update then click ‘OK’.
11. Click on **‘Generate Bitstream’** under ‘PROGRAM AND DEBUG’ in Flow Navigator pane. Click on ‘Yes’ and then ‘OK’.

12. After successful generation of the bitstream for your design, note down the addresses of the different ports of your kernel which will be needed when writing the host code. Export the bitstream file and the handoff file with extensions '.bit' and '.hwh' respectively of your design and upload it to the PYNQ Jupyter notebook and write the host code to validate your design on the board.

13. Get the board running by making necessary connections and run your host code on the notebook. Validate the output of your application at the end.

Chapter 4

Implementation

4.1 Graph Algorithms

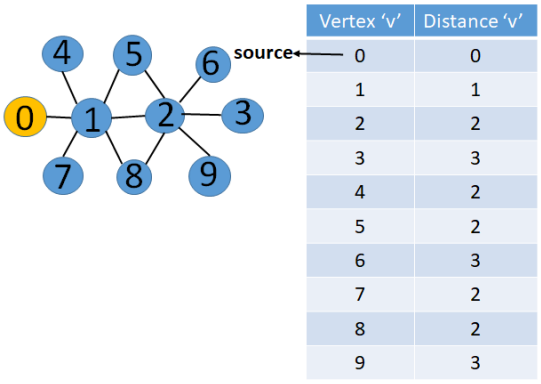
In this section, we will describe the four graph algorithms implemented along with their respective optimizations and partitioning methodologies. We have employed graph partitioning as a solution to process huge real-world graphs on the device because of limited on-chip resources that restrict the entire graphs to be fit on the device.

4.1.1 Single Source Shortest Path

Single source shortest path, briefly abbreviated as SSSP, is a graph traversal algorithm which computes the shortest path distances from a source node to all the other nodes in the entire graph
Caveat: The below algorithm described can be used only with unweighted graphs.

Applications : Road Networks in Google Maps, Routing protocols for packet-routing in the Internet, Telephone networks.

Time complexity = $O(V)$



Vertex 'v'	Distance 'v'
0	0
1	1
2	2
3	3
4	2
5	2
6	3
7	2
8	2
9	3

Figure 4.1: Single Source Shortest Path

Algorithm Description:

In algorithm 1 for SSSP, we maintain three arrays, dist[ ], queue[ ], and visited[ ] that store the distances from the source vertex, traversal order of the graph, and whether the visited/unvisited status of each vertex respectively and each are of size 'V'. col[ ] and row_idx[ ] are the arrays to represent the adjacency matrix of the graph in term of CSR format, where the size of col[ ] is 'E' and size of row_idx[ ] is 'V+1'. We first initialize dist[ ] to ∞ for all nodes except for the source where dist[source] is set to 0. We initialize visited[ ] to 0. We first add the source node to the queue. Variable 'u' denotes the current vertex of the frontier in the queue and variable 'v' denotes all the neighbors of vertex 'u'. Until queue[ ] is empty, we do the following:
If neighbor 'v' of 'u' is not visited, and if the distance of 'v' from source is greater than the sum of distance of 'u' and the edge weight between 'u' and 'v', update the distance of 'v' to be the sum of distance of 'u' and edge weight between 'u' and 'v', add 'v' to the queue, and set visited of 'v' equal to unity. After all the neighbors of 'u' are processed, set visited of 'u' to unity and remove 'u' from the queue.

Algorithm 1 Single Source Shortest Path Algorithm

Input: *Graph*, *Source* ▷ *Graph* representation : CSR
Output: dist[ ] : Distance array containing all distances from source

```
1:  create visited array visited[ ]
2:  create visited array queue[ ]
3:  add source to queue[ ]
4:  initialize dist[ ] ← infinity
5:  initialize visited[ ] ← 0
6:  while queue is not empty do
7:      u ← queue[vertex_at_front]
8:      x ← row_idx[u+1]
9:      for v = row_idx[u] to x do                ▷ iterating over all neighbors of 'u'
10:         if !(visited[col[v]]) and dist[v] > dist[u] + edge_weight(u,v) then
11:             dist[v] ← dist[u] + edge_weight(u,v)        ▷ edge_weight(u,v) = 1
12:             visited[col[v]] ← 1
13:             add col[v] to queue
14:         end if
15:     end for
16:     visited[u] ← 1
17:     remove u from queue[ ]
18:
```

Partitioning Methodologies:

1. ***Multilevel k-way Partitioning Scheme*:** The first partitioning technique we considered for the SSSP algorithm is the **'Multilevel k-way Partitioning Scheme'** where the graph is partitioned into 'k' balanced partitions by minimizing the edge-cuts between the partitions. We used the METIS' serial programs for graph partitioning to achieve partitioning. Partitions obtained by METIS are such that neighborhood vertices are grouped together, which is exactly what we need as SSSP is a graph traversal problem.
   As the vertex ID's are integers from 0 to V-1, we have been using array indices to map different parameters of a vertex, like its distance and visited status, directly. For example, if we require the distance and visited status of vertex 6, then we simply go to the 6th index of the distance array and the visited array to retrieve the required parameters for that particular vertex.

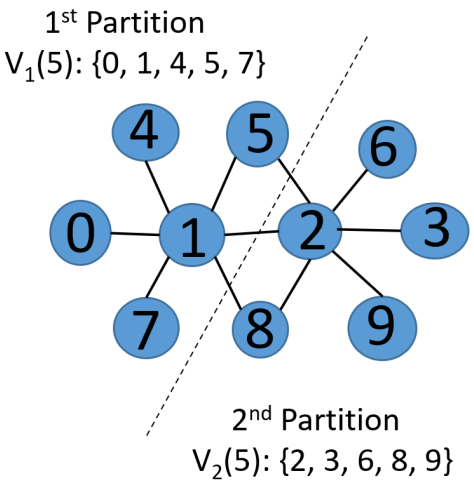


Figure 4.2: Illustration of k-way partitioning output of METIS with k = 2, Number of edge-cuts = 3, Number of vertices in partition 1 and 2 = 5 and 5 respectively

The problem with using the above mentioned partitioning technique is that we lose the direct one-to-one mapping between the vertex IDs and array indices as partitions do not have contiguous vertex IDs. Hence, we need a hash function to map the vertex IDs to the array indices. We have done precisely this using hash tables to store all the vertices in the partition at predetermined locations in the table according to the hash function chosen. The **'keys'** of the hash table will be the vertex IDs in the partitions and the **'values'** will be the distance and visited status associated with its key. We have resolved collisions in the hash table by using **linear probing technique**. The best case lookup time of this method is ***O(1)*** and the worst case is ***O(table_size)***.

The primary pitfall of this method of partitioning is that every time we need distance/visited status of a vertex, we need decode it's location using the hash table. This will lead to reduced performance. Therefore, we have forgone this technique in favor of 'row-wise partitioning scheme' which is described below.

2. ***Row-wise Partitioning Scheme*:** The final partitioning methodology we have employed for the SSSP algorithm is '**Row-wise Partitioning Scheme**' or '**Vertex-wise Partitioning Scheme**' where each partition will have the neighbors of a particular vertex. Therefore, the number of partitions will be equal to the number of vertices in the graph. The partition size of a graph will at most be equal to the maximum degree of the graph.

		0	1	2	3	4	5	6	7	8	9
1st partition	0		1								
2nd partition	1	1		1		1	1		1	1	
3rd partition	2		1		1		1	1		1	1
8th partition	3			1							
4th partition	4		1								
5th partition	5		1	1							
9th partition	6			1							
6th partition	7		1								
7th partition	8		1	1							
10th partition	9			1							

Figure 4.3: Illustration of row-wise partitioning with adjacency matrix representation with V = 10, Number of partitions = 10, Maximum degree of graph = 6

Similar to the unpartitioned algorithm, we will have queue[ ], visited[ ], and dist[ ] each are of size 'V' residing in the off-chip memory. Each partition will have local buffers maintained in the on-chip memory: q[ ], vis[ ], and d[ ]. These are analogous to their global counterparts. With these three arrays, we will also need neighbors[ ] that will store the neighbor vertices of the current partition. The size of each of the local buffers is equal to the 'maximum degree' of the graph.

First, we send all the values of dist[ ] and visited[ ] corresponding to the neighbors of the front vertex in queue[ ] to the kernel. Along with these, we also send the neighbors, distance of source to the current vertex, and the degree of the current vertex to the kernel. The kernel does the necessary computation sends back the updated d[ ], q[ ], and vis[ ] arrays back to the host program. We update the global arrays accordingly and send the next partition. This process takes place until the global queue[ ] is empty. We keep

Algorithm 2 Single Source Shortest Path Algorithm with ROW-WISE PARTITIONING

Input: neighbors[ ], vis[ ], d[ ], udist, degree
Output: q[ ], updated d[ ], updated v[ ], no_of_updates

```
1:  create q[ ]
2:  no_of_updates ← 0
3:  for v = 0 to degree do
4:      #pragma HLS PIPELINE
5:      if !(vis[v]]) and d[v] > udist + 1 then
6:          d[v] ← udist + 1
7:          vis[v] ← 1
8:          add neighbors[v] to q
9:          no_of_updates[v] ← no_of_updates[v] + 1
10:     end if
11: end for=0
```

a track of the number of updates performed in the kernel to facilitate writing the local values of q[ ] to the global queue[ ].

This partitioning technique does not have the complication faced by the first as we traverse each of the local array sequentially and the final writing of local to global array is done by mapping the neighbor array values to the global array indices.

We can use 'PIPELINE' directive in HLS for pipelining the **for** loop in the kernel and can achieve an initiation interval (II) of 1 as there is no carried/inter dependency between any two loop iterations.

4.1.2 PageRank

PageRank is an algorithm for link analysis which assigns a numerical weighting to each node in a hyperlinked set of nodes to measure its relative importance within the set. It is one of most well-known and influential algorithms for computing the relevance of web pages and was used by Google. The equation to calculate PageRank is given below.

$$PR(p_i) = \frac{1-d}{N} + d\left(\sum_{p_j \to p_i} \frac{PR(p_j)}{L(p_j)} + \sum_{p_j \to no\ outlinks} \frac{PR(p_j)}{N}\right) \tag{4.1}$$

The number and quality of pages which link to a particular webpage realises the importance of the page. This is the basic idea behind the PageRank algorithm. It employs the model of an idealized random Web surfer who simply keeps clicking on successive links at random.
The formula used to calculate the PageRank value of a node is shown below:
Time complexity = $O(V^2)$

Applications : Rank web pages, Neuroscience, debugging systems and in predicting road and foot traffic in urban spaces.

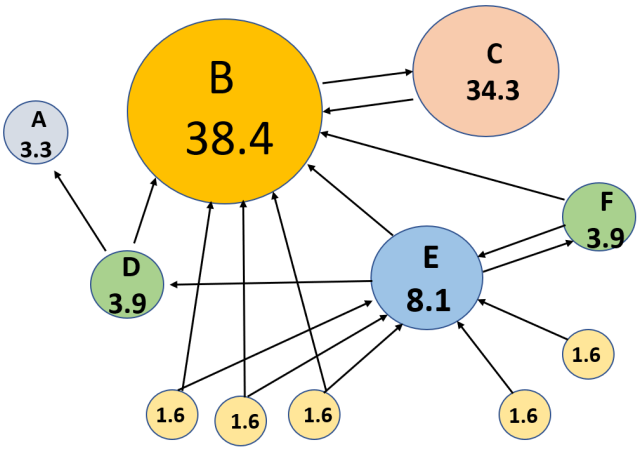


Figure 4.4: PageRank

In the above image, Page B has the highest PageRank value and since it links to Page C, Page C has a higher PageRank value compared to Page E in spite of Page E having many more links towards it. It happens so because Page B which is perceived as important points to Page C, making it important. It is modelled after a person randomly surfing the web and visting random pages. The numerical values represent the likelihood percentage that the page is visited by the web surfer. Damping factor proportionally distributes the likelihood percentages of the surfer

visiting a page among all the pages present in the network. In the absence of this factor, pages with outgoing links would have no PageRank value.

Algorithm 3 PageRank Algorithm

Input: *Graph*, *N* , *iteration* ▷ *Graph* representation : Adjacency Matrix
Output: pg[] : pagerank array

```
d ← 0.85
inlink ← Graph
outlink ← Graph
N ← Graph
for i in N do
    pg[i] ← 1/N
end for
while iteration > 0 do
    dp ← 0
    for i that has no outlinks do
        dp ← dp + d * pg[i] / N
    end for
    for i inlink Graph do
        new_pg[i] ← dp + (1 - d) / N
        for j in inlink[i] do
            new_pg[i] ← new_pg[i] + d * pg[j] / outlink[j]
        end for
    end for
    pg ← new_pg
    iteration ← iteration - 1
end while
```

Algorithm Description:

Initially, all the nodes of the graphs are initialized with the value $\frac{1}{N}$, where N is the number of nodes in the graph. For every iteration, the PageRank values for all the nodes are computed, according to the above formula, and updated accordingly. This process is repeated until the PageRank values of the nodes are converged. The rank of a particular node depends on the PageRank values of the nodes pointing towards that particular node. In case of a dangling page, the PageRank is calulated by dividing its initial PageRank value by the total number of nodes present in the graph. This is how we determine the most important and influential nodes present in the given graph.

Partitioning Methodologies:

Since the sparse matrix vector multiplication is the most computationally intensive step of the PageRank algorithm, it was chosen as the kernel to be run on the FPGA repeatedly for every iteration and partition. The parameter to the kernel include row[ ], col[ ], val[ ], pagerank[ ], and new_pagerank[ ], where row[ ], col[ ] and val[ ] represent the graph in COO format, pagerank[ ] stores the PageRank values of all the nodes computed in the previous iteration and new_pagerank[ ] stores the PageRank vector computed in the present iteration.

Looking into the various partitioning methodologies,

1. ***Row Wise Partitioning Scheme*:** was implemented initially with the help of a variable '**threshold**' which decided the number of rows in each partition. But this type of partitioning was deemed to be inefficient as the number of elements in each partition varied every time, leading to the SPMV kernel iterating variably. It also resulted in the variable sizes of the parameters being passed to the kernel every time. This resulted in an unnecessary over heard whenever the kernel are invoked.

$$\begin{bmatrix} 0 & 0 & \frac{1}{3} & 0 \\ \frac{1}{2} & 0 & \frac{1}{3} & 0 \\ \frac{1}{2} & 0 & 0 & 1 \\ 0 & 1 & \frac{1}{3} & 0 \end{bmatrix}$$

Figure 4.5: Row Wise Partitioning for PageRank

In the above image, a graph consisting of 4 nodes and 7 edges is considered and each row is partitioned (for depiction).

Taking into account all of the above disadvantages, the PageRank algorithm was finally implemented by partitioning according to the succeeding scheme.

2. ***Maximum Elements Partitioning Scheme*:** In this partitioning scheme, the graph is partitioned according to the 'maximum elements' which could fit on the device, again, stored in the variable named 'threshold'. Using this technique of partitioning, unnecessary overhead is reduced whenever the kernel is invoked as the number of iterations of the kernel and the sizes of the parameter to the kernel are fixed for every function call.

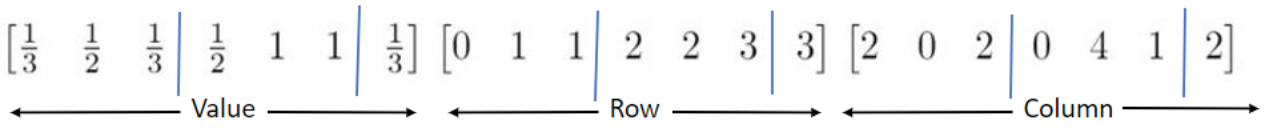


Figure 4.6: Partitioning according to 'max elements'

In the above image, a graph consisting of 4 nodes and 7 edges is considered and the three arrays corresponding to the COO format is divided assuming 'threshold'=3 (depicted by the blue lines).

The only point to be considered while implementing this type of partitioning is that the graph must be partitioned in such a way that every partition must have equal number of elements. In order words, the value stored in the **'threshold'** must be a factor of the edges present in the graph. In case the above condition is not possible this type of partitioning would still work but the iterations of the kernel and the parameters would vary for the very last partition as it would contain the remaining elements which did not fit in the other equal sized partitions.

The only drawback of this partitioning methodology is that the PageRank vector must be globally available to all the partitions of the graph and hence limits large graphs from being verified by the device.

4.1.3 Breadth First Search

BFS is a graph traversal algorithm which starts to traverse from a selected node, graph layer wise, thus exploring the neighbour nodes. We then move towards the next-level neighbour nodes.

Time complexity = $O(V)$

Applications : Finding all neighbors in Peer to Peer Networks, Finding all neighboring locations in GPS Navigation System, Broadcasting packets in Networks.

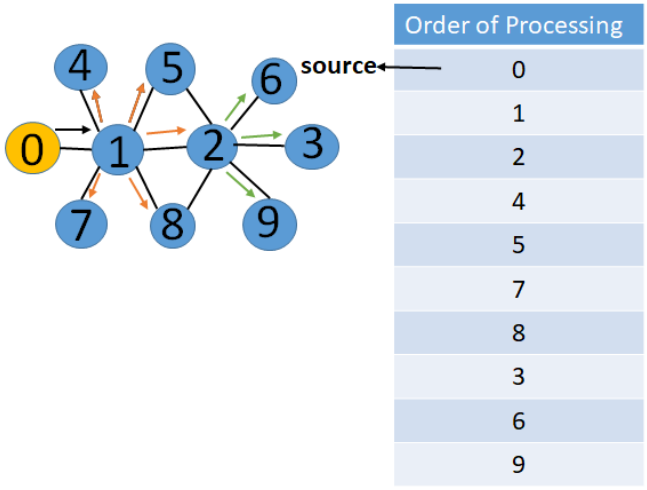


Figure 4.7: Breadth First Search Traversal

Algorithm Description:

For Breadth First Search, we maintain two arrays queue[ ] and visited[ ] that stores traversal order of the graph, and whether the visited/unvisited status of each vertex respectively.col[ ] and row_idx[ ] are the arrays to represent the adjacency matrix of the graph in terms of CSR format, where the size of col[ ] is 'E' and size of row_idx[ ] is 'V+1'. We initialize visited[ ] to 0, as no vertex is encountered yet. We first add the source node to the queue. Variable 'u' denotes the current vertex of the frontier in the queue and variable 'v' denotes all the neighbors of vertex 'u'. Until queue[ ] is empty, we do the following:If neighbor 'v' of 'u' is not visited,add 'v' to the queue, and set visited of 'v' equal to unity. After all the neighbors of 'u' are processed, set visited of 'u' to unity.After all the neighbors of'u' are processed, set visited of 'u' to unity and remove 'u' from the queue.

Algorithm 4 Breadth First Search Algorithm

Input: *Graph*, *Source* ▷ *Graph* representation : CSR
Output: queue[] : BFS traversal array

```
1:  create visited array vis[]
2:  create traversal array queue[ ]
3:  initialize visited[ ] ← 0
4:  add source to queue
5:  while queue is not empty do
6:      u ← queue[vertex_at_front]
7:      x ← row_idx[u+1]
8:      for v = row_idx[u] to x do                ▷ iterating over all neighbors of 'u'
9:          if !(vis[col[v]]) then
10:             vis[col[v]] ← 1
11:             add col[v] to queue
12:         end if
13:     end for
14:     visited[u] ← 1
15:     remove u from queue[ ]
```

Partitioning Methodologies:

1. ***Multilevel k-way Partitioning Scheme*:** The first partitioning technique we considered for the BFS algorithm is the **'Multilevel k-way Partitioning Scheme'** where the graph is partitioned into 'k' balanced partitions by minimizing the edge-cuts between the partitions. We used the METIS' serial programs for graph partitioning to achieve partitioning. Partitions obtained by METIS are such that neighborhood vertices are grouped together, which is exactly what we need as BFS is a graph traversal problem. As the vertex ID's are integers from 0 to V-1, we have been using array indices to map different parameters of a vertex, like its distance and visited status, directly.

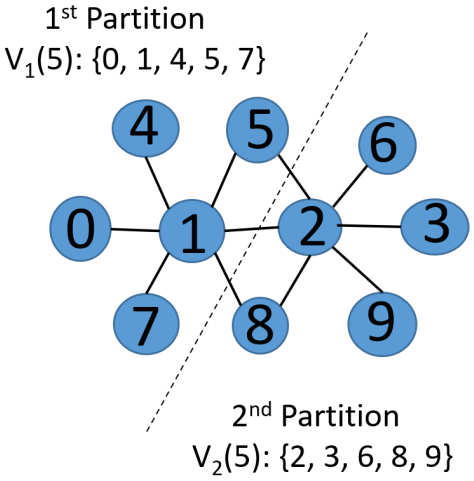


Figure 4.8: Illustration of k-way partitioning output of METIS with k = 2, Number of edge-cuts = 3, Number of vertices in partition 1 and 2 = 5 and 5 respectively

The problem with using the above mentioned partitioning technique is that we lose the direct one-to-one mapping between the vertex IDs and array indices as partitions do not have contiguous vertex IDs. Hence, we need a hash function to map the vertex IDs to the array indices. We have done precisely this by utilizing hash tables to store all the vertices in the partition at predetermined locations in the table according to the hash function chosen. The **'keys'** of the hash table will be the vertex IDs in the partitions and the **'values'** will be the visited status associated with its key. We have resolved collisions in the hash table by using **linear probing technique**. The best case lookup time of this method is ***O(1)*** and the worst case is ***O(table_size)***.

The primary pitfall of this method of partitioning is that every time we need visited status of a vertex, we need decode it's location using the hash table. This will lead to reduced performance. Therefore, we have forgone this technique in favor of 'row-wise partitioning scheme' which is described below.

2. ***Row-wise Partitioning Scheme*:** The final partitioning methodology we have employed for the BFS algorithm is '**row wise partitioning**'where each partition will have the neighbors of a particular vertex and we use CSR format of sparst matrix representation. Therefore, the number of partitions will be equal to the number of vertices in the graph. The partition size of a graph will at most be equal to the maximum degree of the graph.

		0	1	2	3	4	5	6	7	8	9
1st partition	0		1								
2nd partition	1	1		1		1	1		1	1	
3rd partition	2		1		1		1	1		1	1
8th partition	3			1							
4th partition	4		1								
5th partition	5		1	1							
9th partition	6			1							
6th partition	7		1								
7th partition	8		1	1							
10th partition	9			1							

Figure 4.9: Illustration of row-wise partitioning with adjacency matrix representation with V = 10, Number of partitions = 10, Maximum degree of graph = 6

Similar to the unpartitioned algorithm, we will have queue[ ] and visited[] each are of size 'V' residing in the off-chip memory. Each partition will have local buffers maintained in the on-chip memory: q[ ] and vis[ ].These are

analogous to their global counterparts. With these two arrays, we will also need neighbors[ ] that will store the neighbor vertices of the current partition. The size of each of the local buffers is equal to the 'maximum degree' of the graph. First, we send all the values of visited[ ] corresponding to the neighbors

Algorithm 5 Breadth First Search Algorithm with ROW-WISE PARTITIONING

Input: neighbors[ ], vis[ ], degree
Output: q[ ], updated v[ ], no_of_updates

```
1:  create q[ ]
2:  no_of_updates ← 0
3:  for v = 0 to degree do
4:      #pragma HLS PIPELINE
5:      if !(vis[v]]) then
6:          vis[v] ← 1
7:          add neighbors[v] to q
8:          no_of_updates[v] ← no_of_updates[v] + 1
9:      end if
10: end for=0
```

of the front vertex in queue[ ] to the kernel. Along with these, we also send the neighbors, and the degree of the current vertex to the kernel. The kernel does the necessary computation sends back the updated q[ ], and vis[ ] arrays back to the host program. We update the global arrays accordingly and send the next partition. This process takes place until the global queue[ ] is empty. We keep a track of the number of updates performed in the kernel to facilitate writing the local values of q[ ] to the global queue[ ].

We can use 'PIPELINE' directive in HLS for pipelining the **for** loop in our kernel and can achieve an initiation interval (II) of 1 as there is no carried/inter dependency between any two loop iterations.

4.1.4 Depth First Search

DFS is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root or source node and explores as far as possible along each branch before backtracking.

Time complexity = $O(V * E)$

Applications : To find Strongly connected components and Weakly connected components, Detecting cycles in a graph, Finding path between two vertices.

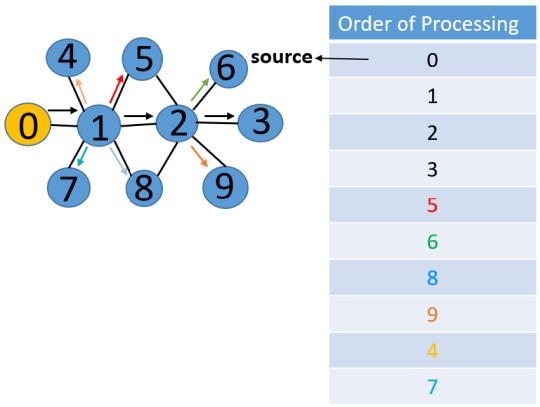


Figure 4.10: Depth First Search Traversal

Algorithm 6 Depth First Search

Input: *Graph*, *Source*
Output: dfs[vertex] : DFS traversal array ▷ *Graph* representation : COO

```
Create stack[vertex] ← 0
Create vis[vertex] ← 0
for all vertices of the graph do
    s ← stack[top-1]
    vis[s] ← 1
     dfs[vertex] ← s
    for all edges in the graph do
        if row[edge] == s and vis[col[edge]] == 0 then
            stack[top] ← col[edge]
        end if
    end for
end for
```

Algorithm Description:

For Depth First Search, we create three arrays stack[ ],vis[ ] and d[ ] we start traversing from source vertex and push all its neighbor vertices into the stack. Pop a vertex from stack to select the next vertex to visit and push all its neighbor nodes into the stack.To avoid processing a vertex more than once, we use a Boolean visited array.col[ ] and row[ ] are the arrays to represent adjacency matrix of the graph in terms of COO format, where the size of col[ ] and row[ ] is 'E'.

We initialize vis[ ] to 0, as no vertex is encountered yet, we first add the source vertex to the stack. Variable 's' denotes the current vertex, add s to stack and make s as visited, add s to d[ ] . Until stack[ ]is empty, we do the following:If neighbors of 's' are not visited,add them to the stack, and set visited status of neighbors equal to unity.

4.2 Experimental Setup

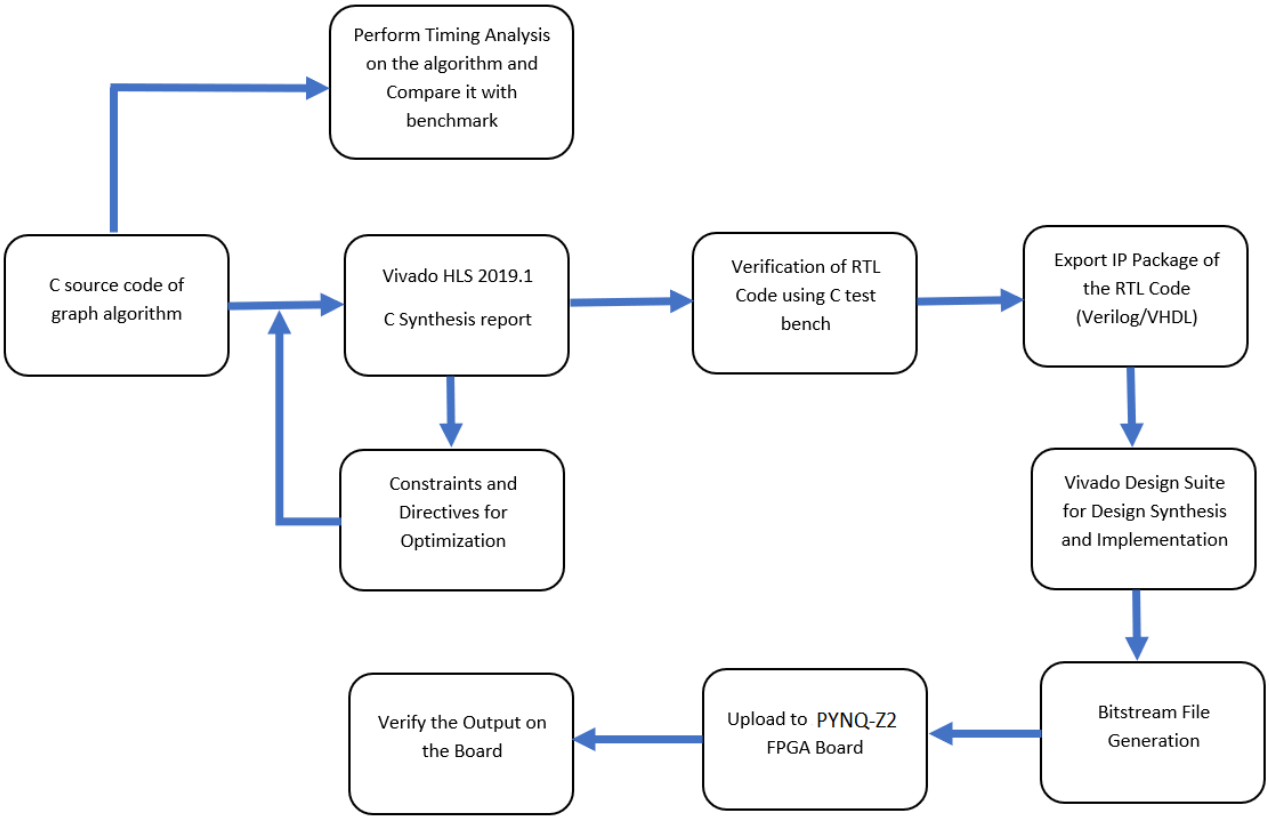


Figure 4.11: Experimental Setup

1. The algorithm is implemented in Vivado HLS. Optimization directives are applied appropriately to improve performance, Initiation Interval (II), throughput, and resource utilization of the design. C synthesis is performed to generate the RTL files.
2. C/RTL Cosimulation and Wave Debug are performed to get the exact latencies for the kernel and debug the design at signal-level if necessary.
3. The IP of the HLS implementation is exported and Verilog synthesis, place and route is performed to generate the report for the exact clock period achieved by the implementation.
4. In the Vivado Design Suite using the IP Integrator, a new block design using the exported IP with the Zynq7 Processing System as the host is created. HDL Wrapper for the block design is created.

5. Synthesis and Implementation for the design is performed and the bitstream is generated. The necessary files, i.e., the bitstream file (.bit) and the handoff file (.hwh) are exported for validating the design on the hardware. Port addresses for your IP is noted.
6. The image for PYNQ-Z2 board is flashed and the board is booted up. Host code for validating the design is written in Jupyter notebook and the functionality of the your application kernel is verified on the board.
7. Necessary data from reports generated post-implementation of the block design are noted for further analysis.

Chapter 5

Result Analysis

5.1 Execution timings

In order to standardize evaluations, we used the high-performance reference implementation of a few graph algorithms (namely BFS, SSSP, and PageRank) in the GAP Benchmark Suite as the baseline to reliably compare our results. We do not include the performance comparison for DFS because the GAP Benchmark does not provide it in its repository.
(The GAP Benchmark Suite is one of the outcomes of the Graph Algorithm Platform (GAP) Project by a group from UC, Berkeley.)

Benchmark: High-performance reference code executed on CPU
Our Code: Code written and optimised in HLS and executed for PYNQ-Z2 FPGA
Graph Size: 4720 nodes, 27444 edges

Algorithm	Latency	Clock (ns)	Execution Time (ms)	GAP Benchmark Time (ms)
BFS	311520	6.319	1.968	9.808
SSSP	457840	7.282	3.33	67.215
PageRank	2648400	8.644	22.89	1.256
DFS	335354529	8.373	2808	-

Table 5.1: Comparison of execution times of graph algorithms in milliseconds(ms) with GAP Benchmark

System specifications:
Processor : AMD A9-9420 @ 3.000GHz. CPU
OS : KDE neon
RAM : 8.00GB

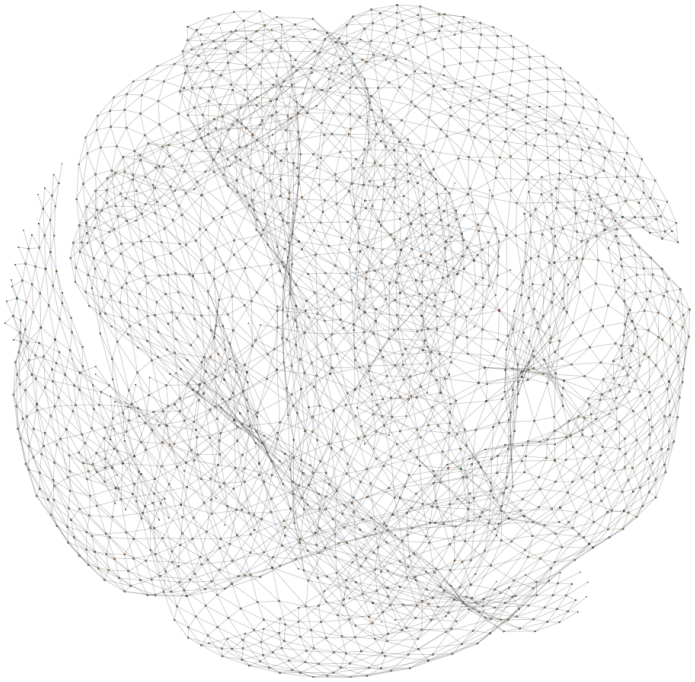


Figure 5.1: Graph used in timing analysis; 4720 nodes, 27444 edges.

Shown below is the plot of Execution Time of a Graph Algorithm in milliseconds(ms) on the y-axis and the corresponding Graph Algorithm on the x-axis.

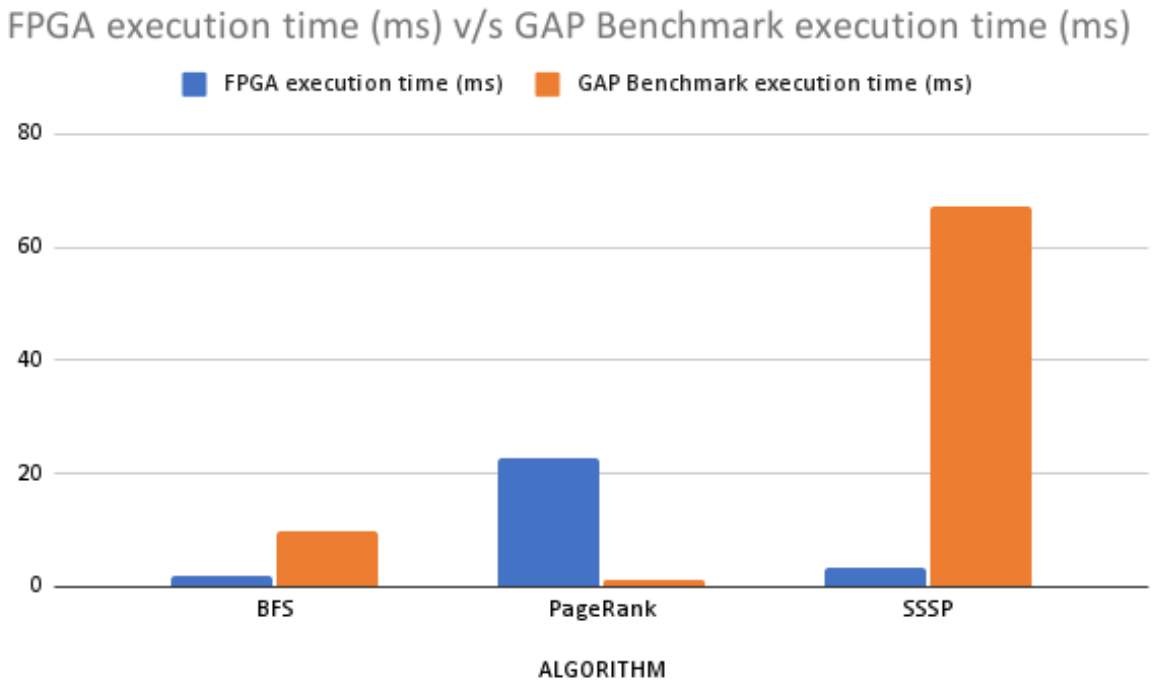


Figure 5.2: FPGA execution time (ms) versus GAP Benchmark execution time (ms)

5.2 Post-Implementation Results of the Algorithms

5.2.1 Block Design

As we have maintained a standard interface across all the kernels, the block designs for all our applications is similar where only change is that the IP exported from the HLS is individual to that algorithm. We have used two bundled master AXI interfaces for I/O operations between the IP and the Zynq PS. Figure 5.3 shows the block design for the SSSP algorithm.

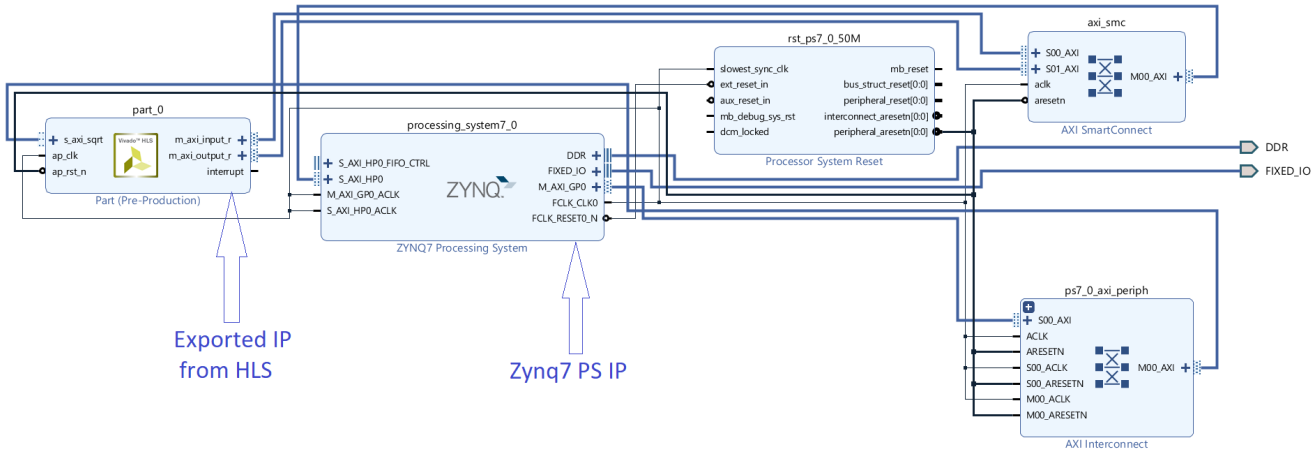


Figure 5.3: Block Design for SSSP

5.2.2 Utilization Reports

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	5181	53200	9.74
LUTRAM	833	17400	4.79
FF	6911	106400	6.50
BRAM	1.50	140	1.07
BUFG	1	32	3.13

Figure 5.4: Utilization Report for SSSP

Utilization | Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	2844	53200	5.35
LUTRAM	349	17400	2.01
FF	3832	106400	3.60
BRAM	1	140	0.71
BUFG	1	32	3.13

Figure 5.5: Utilization Report for PageRank

Utilization | Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	5079	53200	9.55
LUTRAM	801	17400	4.60
FF	6758	106400	6.35
BRAM	1.50	140	1.07
BUFG	1	32	3.13

Figure 5.6: Utilization Report for BFS

Utilization | Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	2946	53200	5.54
LUTRAM	347	17400	1.99
FF	3865	106400	3.63
BRAM	97.50	140	69.64
BUFG	1	32	3.13

Figure 5.7: Utilization Report for DFS

5.2.3 Power Analysis

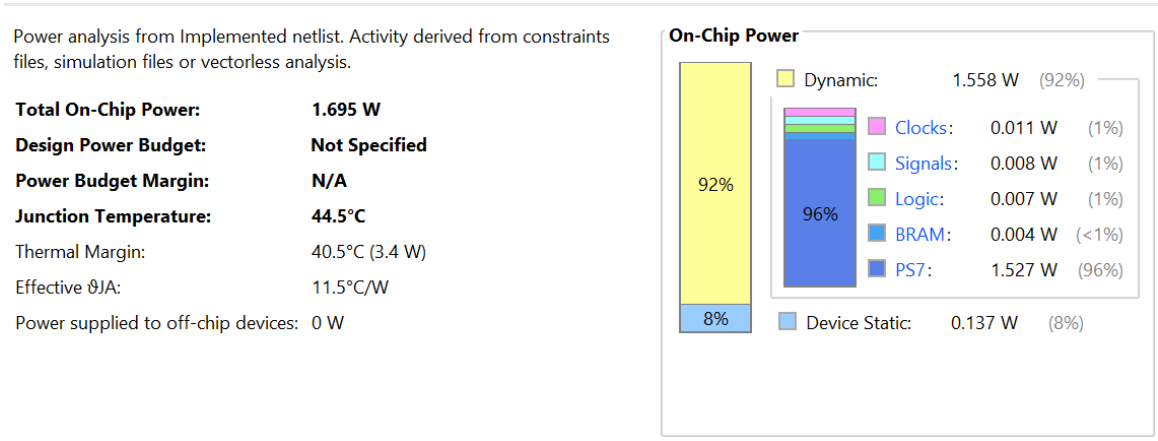


Figure 5.8: Power Analysis of SSSP

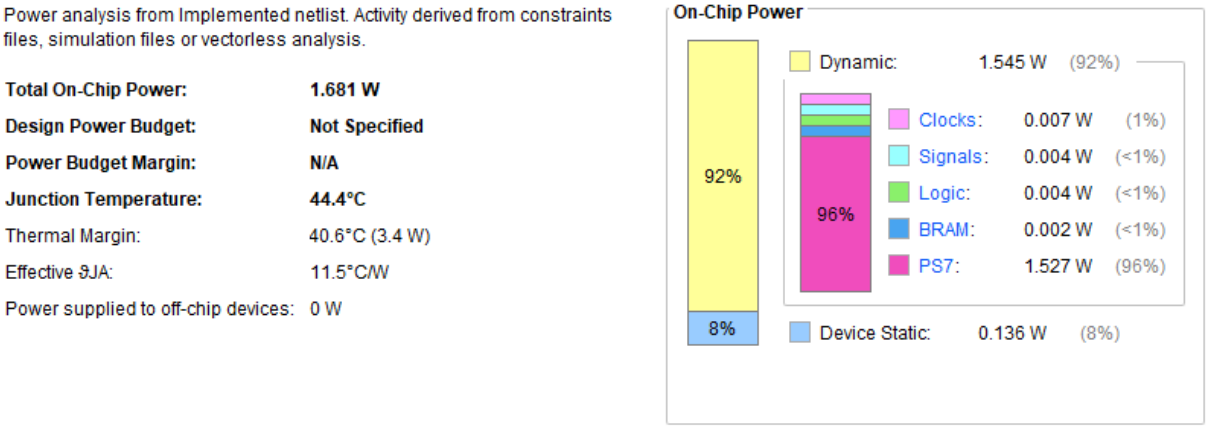


Figure 5.9: Power Analysis of PageRank

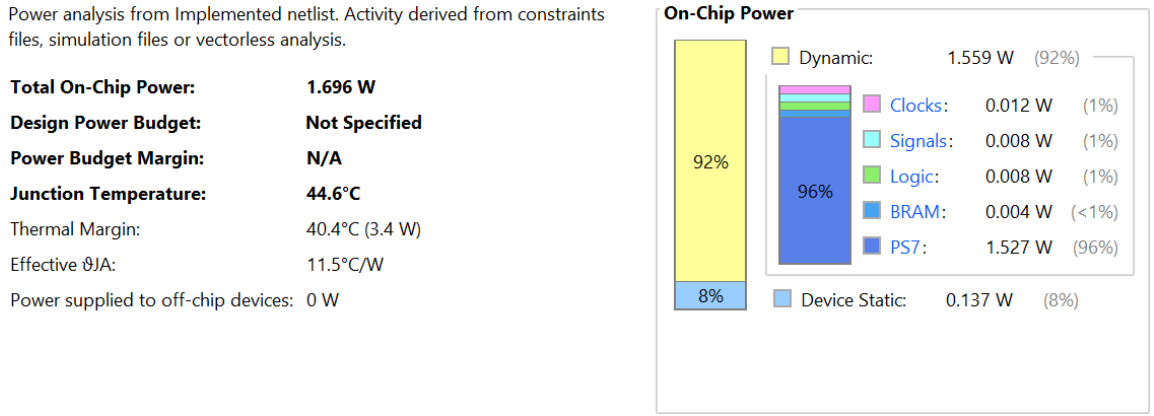


Figure 5.10: Power Analysis of BFS

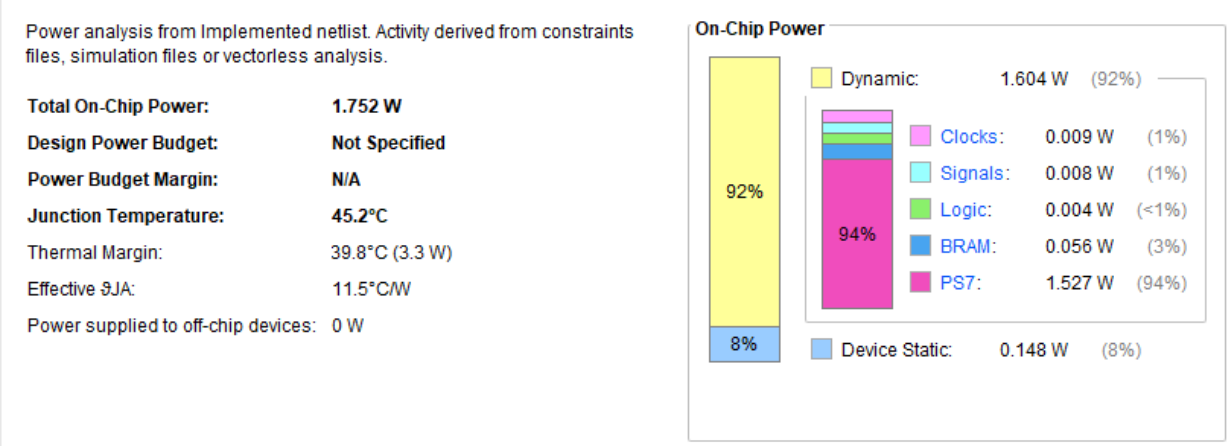


Figure 5.11: Power Analysis of DFS

Algorithm	Energy consumed on FPGA (J)	Energy consumed on CPU (J)
BFS	0.003338	0.0193
SSSP	0.00565	0.132
PageRank	0.03847	0.002267
DFS	4.919	-

Table 5.2: Energy Consumption on FPGA and CPU

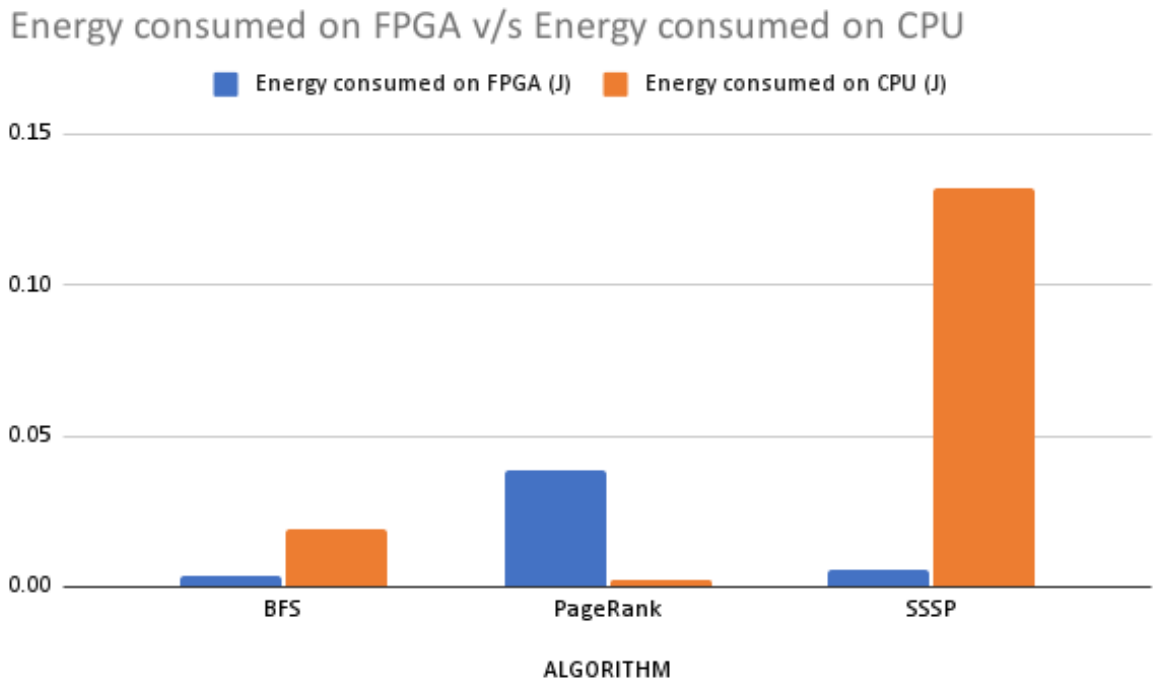


Figure 5.12: Comparison of Energy Consumption of each algorithm on FPGA and CPU

5.2.4 Post-Implementation Designs

The below figures show how the logic for each algorithm is mapped to the device resources.

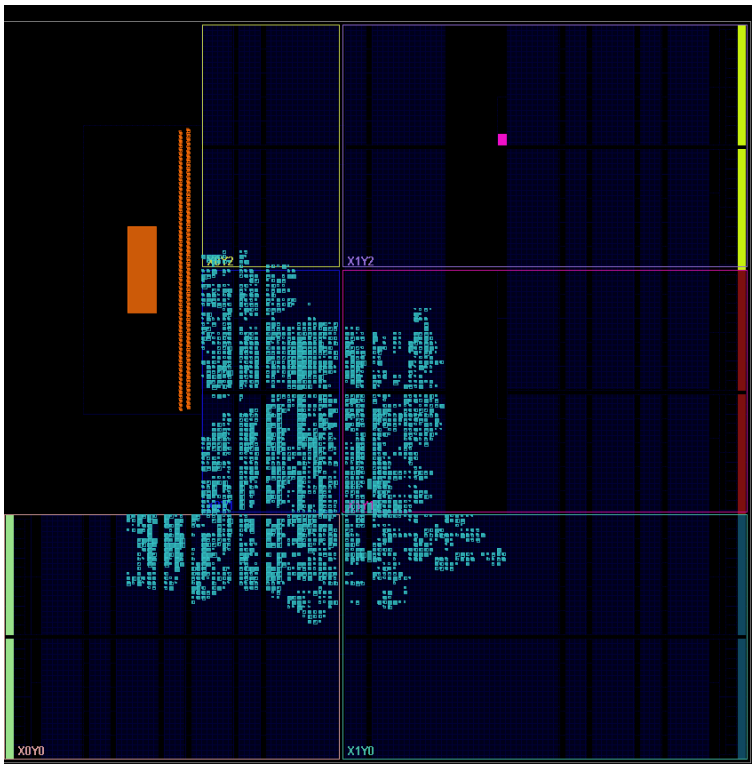


Figure 5.13: Design for SSSP

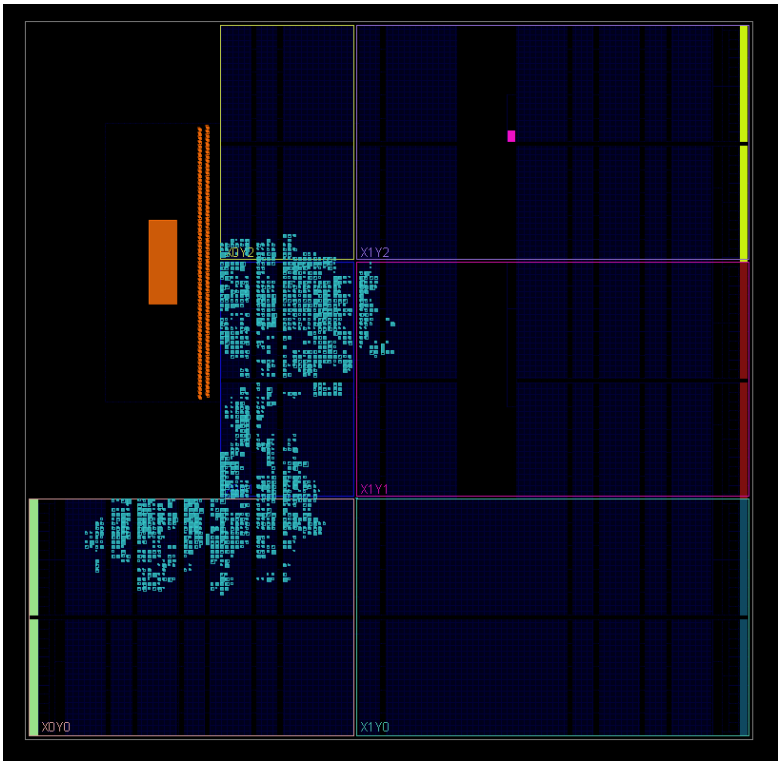


Figure 5.14: Design for PageRank

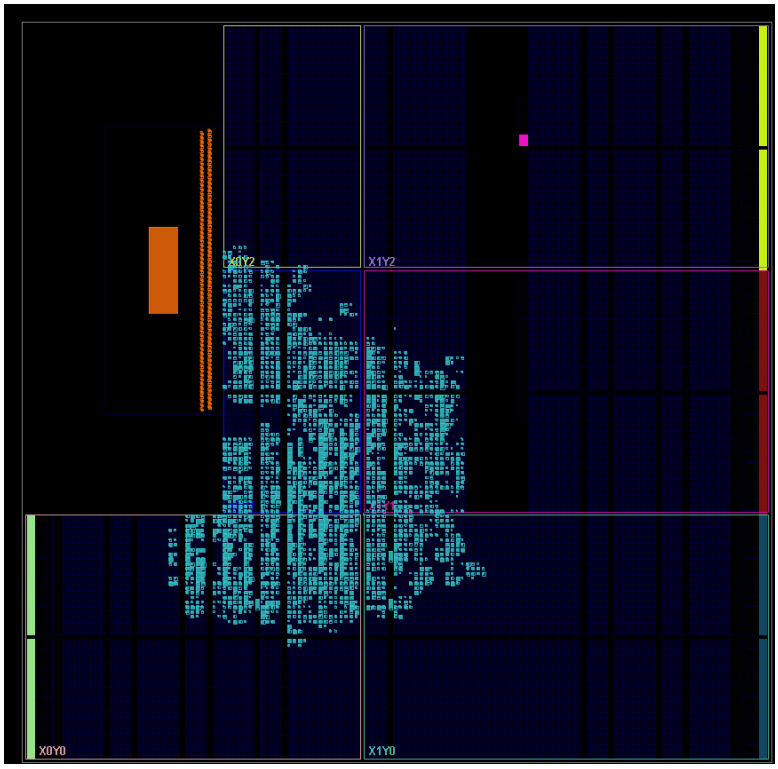


Figure 5.15: Design for BFS

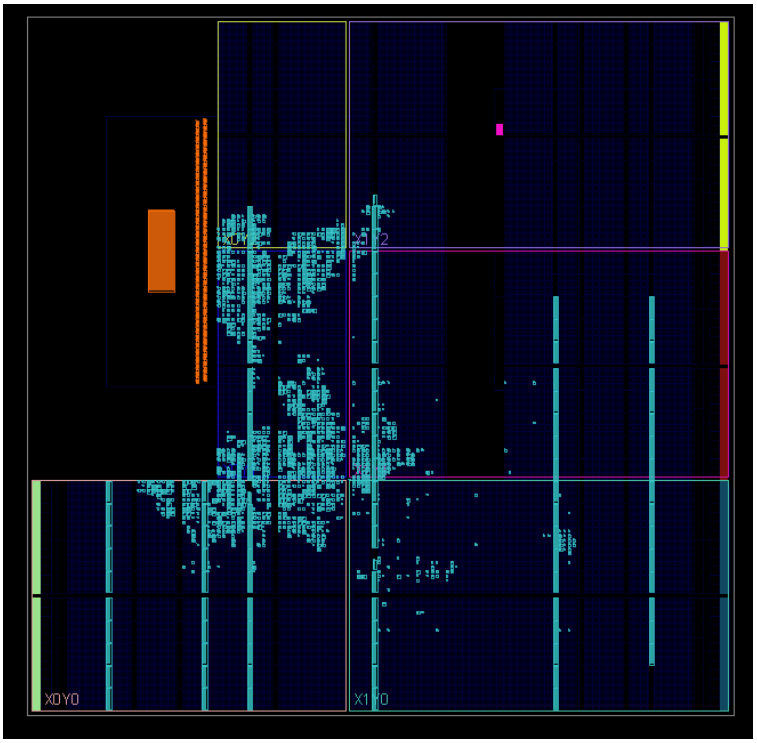


Figure 5.16: Design for DFS

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

We have implemented a custom-made FPGA accelerator for each of the four graph algorithms discussed in the previous sections. Because of the constraint on the on-chip memory, we have considered and performed graph partitioning to facilitate processing of large graphs on the resource-constrained devices. We have observed that our FPGA accelerator fares better than its CPU counterpart for two algorithms, namely Breadth-First Search, Single Source Shortest Path, but doesn't do well for PageRank. This could be attributed to the fact that PageRank is an iterative algorithm and impacts the pagerank values which in turn affects performance.

The partitioning methodology used in PageRank, i.e., maximum elements partitioning (MEP), could be extended for SSSP and BFS when all the neighbors cannot be accommodated on the device. Therefore, we could then use MEP with respect to the number of neighbors sent to the kernel processing the partitions.

The codes used in this project are available on GitHub at the following link.

6.2 Future Scope

The future scope of our project would be to explore unique solutions for accelerating PageRank without compromising on the precision of the results, consider more graph algorithms that could benefit from FPGA acceleration, and also to integrate all the different kernels for different algorithms on one device.

References

[1] *Graph Processing on FPGAs: Taxonomy, Survey, Challenges*; Maciej Besta, Dimitri Standojevic *et al*.

[2] *Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite*; Ariful Azad, Mohsen Mahmoudi Aznavehy, Scott Beamer *et al*.

[3] *A reduced-precision streaming SpMV architecture for Personalized PageRank on FPGA*; Alberto Parravicini, Francesco Sgherzi and Marco D. Santambrogio.

[4] *A Study of Partitioning Policies for Graph Analytics on Largescale Distributed Platforms*;Gurbinder Gill, Roshan Dathathri, Loc Hoang and Keshav Pingali.

[5] *Accelerating PageRank using Partition-Centric Processing*;Kartik Lakhotia, Rajgopal Kannan and Viktor Prasanna.

[6] *An FPGA Framework for Edge-Centric Graph Processing*; Shijie Zhou, Rajgopal Kannan, Hanqing Zeng and Viktor K. Prasanna.

[7] *HitGraph: High-throughput Graph Processing Framework on FPGA*; Shijie Zhou , Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman and Qing Wu.

[8] *Optimizing Memory Performance for FPGA Implementation of PageRank*; Shijie Zhou, Charalampos Chelmis and Viktor K. Prasanna.

[9] *The PageRank Citation Ranking: Bringing Order to the Web*; Lawrence Page, Sergey Brin, Rajeev Motwani and Terry Winograd.

[10] *ThunderGP: HLS-based Graph Processing Framework on FPGAs*; Xinyu Chen1, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong and Deming Chen.

[11] Vivado HLS – Tips and Tricks by Frédéric Rivoallon.

[12] Vitis High-Level Synthesis User Guide by Xilinx

[13] Vitis Unified Software Platform Documentation by Xilinx

[14] Vivado Design Suite User Guide by Xilinx

[15] METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering
http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

[16] Parallel Programming for FPGAs by Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer.

[17] sbeamer/gapbs:GAPBenchmarkSuite

[18] https://github.com/purtroppo/PageRank

[19] https://en.wikipedia.org/wiki/High-level_synthesis

[20] https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[21] www.ccs.neu.edu/home/daikeshi/notes/PageRank.pdf

[22] https://en.wikipedia.org/wiki/Centrality

[23] https://ocw.mit.edu

[24] https://en.wikipedia.org/wiki/PageRank

[25] sbeamer/gapbs:GAPBenchmarkSuite

[26] https://github.com/purtroppo/PageRank

[27] http://kastner.ucsd.edu/hlsbook/

[28] https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

[29] https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

[30] https://en.wikipedia.org/wiki/High-level_synthesis

[31] https://https://www.geeksforgeeks.org/

[32] http://networkrepository.com/3elt.php#panel-body

[33] https://hardwarebee.com/understanding-fpga-programming-and-design-flow/

Appendix A

Review Process

A.1 Phases of Review

Project was evaluated at phases with major checkpoints as follows :

Pre-Review : ***Defining Problem Statement and Feasibility analysis Review***

1. Understanding of the problem statement.
2. Technical understanding of domain.
3. Identification of differentiating features.
4. Feasibility of conversion to product or paper.

1st Review : ***Literature survey and Project plan Review***

1. Clarity in understanding of the problem/project.
2. Completion of Literature Survey.
3. Identification of sub-blocks and their interaction.
4. Timeline for completion of project using Gantt chart.

2nd Review : ***Module level design and Test Plan Review***

1. Detail design of each module.
2. Integration and module test plan.
3. Availability status of required Hardware and Software components.
4. 20% Completion of Block/Sub module implementation.

3rd Review : ***Project Progress Review***

1. Adherences to project plan.

2. Completion of module interaction interface design.
3. 50% Module level implementation.
4. Demonstration of completed modules using primitive interfaces

4th Review : ***Project Progress and Finishing plan Review***

1. 70% Module level implementation
2. Presentation of completed test data as per test plan.
3. Packaging plan.
4. Final demonstration plan.

5th Review : ***Module completion and Integration Review***

1. 100% Module implementation.
2. Completion of Integration.
3. Presentation of test data as per test plan.
4. Adherences to project plan.

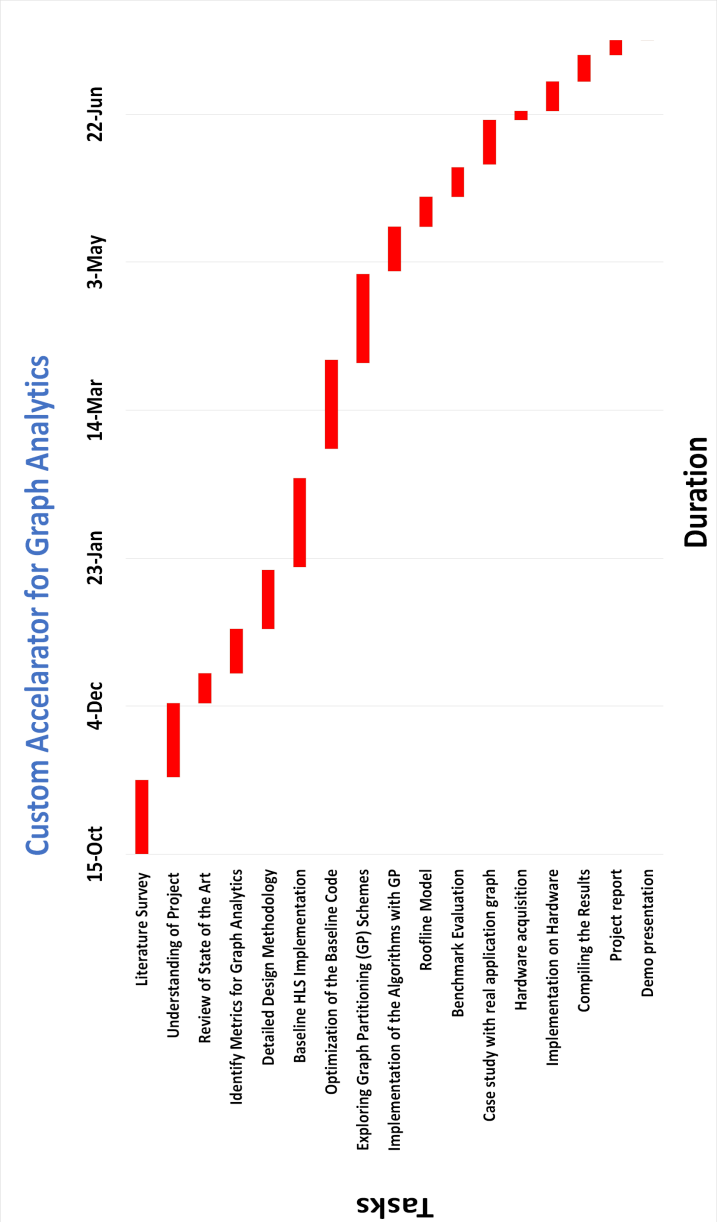
6th Review : ***Demonstration Review***

1. Quality of packaged full demo presentation.
2. Integration test data as per test plan.
3. Quality of documentation made across project.
4. Group presentation / communication skill.

Appendix B

Project Planning

B.1 Gantt Chart



Appendix C

Work Division

C.1 Work Done in 7th Semester

- Literature Survey.
- Software implementation of Graph Algorithms in C code.
- Evaluated and compared the C source code with corresponding reference code in the GAP Benchmark Suite.

C.2 Work Done in 8th Semester

- Learning about HLS and its optimisation methodologies.
- Implementing and optimizing the respective graph algorithms in Vivado HLS tool.
- Exploring and implementing graph algorithms with graph partitioning schemes.
- Running and verifying the functionalities of the algorithms on the FPGA board.