OPENCL-BASED EFFICIENT HLS IMPLEMENTATION OF ITERATIVE GRAPH ALGORITHMS ON FPGA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Kenan Çağrı Hırlak
December 2020

By Kenan Çağrı Hırlak
December 2020

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Özcan Öztürk(Advisor)

Süleyman Tosun

Uğur Güdükbay

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

OPENCL-BASED EFFICIENT HLS IMPLEMENTATION OF ITERATIVE GRAPH ALGORITHMS ON FPGA

Kenan Çağrı Hırlak
M.S. in Computer Engineering
Advisor: Özcan Öztürk
December 2020

The emergence of CPU-FPGA hybrid architectures creates a demand for high abstraction programming tools such as High-Level Synthesis (HLS). HLS handles most of the FPGA development tasks automatically, thus freeing up programmers to create applications effortlessly on FPGAs with familiar programming languages. However, HLS often trades speed for convenience, which makes it a poor choice when it comes to applications in which computational performance is a crucial requirement, such as graph algorithms. In the scope of iterative graph algorithms, we developed custom HLS-based optimizations. Specifically, we applied these on PageRank (PR), Breadth-First Search (BFS), and Connected Components (CC) algorithms so that they can be synthesized in a performant way by HLS tools. We observed that well-pipelined OpenCL kernels can provide up to three times speedups on the Intel Xeon-FPGA architecture compared to CPU implementations. We optimized the traversal of vertices for pipelining to execute applications faster. Furthermore, our approach relies on the HLS workflow to make it effortless for the programmer.

Keywords: Graph Algorithms, High Level Synthesis (HLS), Field Programmable Gate Array (FPGA), PageRank (PR), Breadth First Search (BFS), Connected Components (CC).

ÖZET

YİNELEMELİ ÇİZGE ALGORİTMALARININ FPGA ÜZERİNDE OPENCL İLE ETKİN HLS UYGULAMASI

Kenan Çağrı Hırlak
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Özcan Öztürk
Aralık 2020

İşlemci ve Alanda Programlanabilir Kapı Dizisi (CPU-FPGA) hibrit mimarilerinin yaygınlaşmasıyla Yüksek Seviyeli Sentez (High-Level Synthesis-HLS) gibi soyut programlama yöntemlerine olan ilgi de artmıştır. Bu yöntemler FPGA programlamak için yapılması gereken bir çok işi programcının üzerinden almakta ve otomatikleştirmektedir. Böylece programcı alışık olduğu dil ve yöntemleri kullanarak FPGA için kod geliştirebilmektedir. Ancak bu iş akışı çoğu kez zahmetsiz olma kaygısıyla hızlı çalışmayı göz ardı edebilmekte ve bu nedenle çizge algoritmaları gibi yüksek performans gerektiren uygulamalarda başarısız kalmaktadır. Bu problemi çözmek maksadıyla yinelemeli grafik algoritmalar kapsamında, özel Yüksek Seviyeli Sentez (HLS) tabanlı optimizasyonlar geliştirdik. Özellikle, Sayfa Sıralama, Sığ Öncelikli Arama ve Bağlantılı Bileşenler algoritmalarını HLS ile hızlı bir donanımın sentezlenebilmesi için optimize ettik. FPGA üzerinde gerçekleştirilen verimli bir boru hattına sahip OpenCL çekirdeklerinin CPU üzerinde çalışan uygulamalara kıyasla üç kata kadar daha hızlı çalışabileceğini gösterdik. Çizgeleri boru hattına uygun biçimde kateden bir yöntem geliştirdik. Buna ek olarak, yöntemlerimizi programcıların kolayca kullanabilmesi için Yüksek Seviyeli Sentez (HLS) iş akışına uygun olarak yaptık.

Anahtar sözcükler: Çizge Algoritmaları, Yüksek Seviyeli Sentez (HLS), Alanda Programlanabilir Kapı Dizisi (FPGA), Sayfa Sıralama (PR), Sığ Öncelikli Arama (BFS), Bağlantılı Bileşenler (CC).

Acknowledgement

This work has been supported in part by a grant from Türk Havacılık ve Uzay Sanayii A.Ş. and by Scientific and Technological Research Council of Turkey (TUBITAK) 1001 program through the EEEAG 119E559 project.

I would like to thank my advisor Özcan Öztürk for his patience and support.

I would like to thank the members of the committee, Süleyman Tosun and Uğur Güdükbay, for sparing the time to evaluate this work.

Finally, I must express my very profound gratitude to my family and friends.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Google Search Engine has indexed 130 trillion web pages, as reported in the "How Google Search Works" blog [1]. In every minute passed, a total of 300 hours of video are uploaded to Youtube [2]. A total of 630.000 companies are publicly traded throughout the world [3]. As days go by, the integration of big data into our lives gets only deeper. Graph algorithms are developed and optimized heavily for managing these kinds of magnitudes. PageRank is an example of this upon which Google Search Engine was established [1].

If graph algorithms are the software of this grand scheme, then what is the hardware? There are many options to choose from such as CPUs, GPUs, or newly emerging NPUs. To this end, Intel developed a hybrid architecture that consists of server-class Xeon multicore processors and 20nm state-of-the-art Arria 10 FPGAs. This architecture provides a full-stack of communication framework between main memory, CPU, and FPGA that enables the implementation of performant graph applications that exploits FPGA's adaptivity [4].

The combination of a good software foundation and powerful hardware is a good start but lacking without programmers. FPGAs are notoriously hard to program since programmers need to deal with the issues such as clocks, routing, pipelining, and memory, etc. High-Level Synthesis (HLS) is a rising trend that

rescues programmers from these difficulties and saves their precious time. But it is not without its disadvantages, especially when it comes to customizability and speed.

1.1 Objective of the Thesis

The aim of this thesis is first to survey implementation options for graph algorithms on HLS workflows and propose solutions to improve performance. We use OpenCL to program FPGA kernels, which is a well-known programming language, as opposed to HDLs, which are obscure for most programmers.

There is a wide range of implementations of graph algorithms that offer different types of optimizations and speed improvements. But, we are not free to utilize them to our liking since one of the most crucial factors for achieving speed is to develop kernels in such a way that HLS would be able to pipeline them effectively. Thus, we need to select a subset of these methodologies that can be implemented in OpenCL and suitable for HLS to synthesize. In light of these, we aim to propose techniques to produce fast OpenCL kernels that are optimized for HLS workflows. Our optimizations will target the behavioral specification part of the HLS flow [5].

1.2 Organization of the Thesis

Following the introduction, Chapter 2 presents the proposed optimizations for graph algorithms in notable related works. The first section will be about algorithmic optimizations that increase the speed independently from hardware and implementation methods. The second section will present the optimizations specifically intended for High-Level Synthesis workflows. In Chapter 3, we will give general information about graph algorithms that we implemented. Then, we describe the target hardware and the workflow of the implementation. We

briefly visit some concepts required for this workflow. Chapter 4 provides the implementation details about how we program and execute the graph applications. We aim to give a programming framework for fast OpenCL kernels for HLS implementations. In Chapter 5, we present the implementation details of the PR, BFS, and CC algorithms. We elaborate on the optimizations that we perform with their code snippets. Chapter 6 presents the experimental setup and the results of the experiments. In Chapter 7, we interpret the results and future research opportunities. Finally, Chapter 8 summarizes and concludes the thesis.

Chapter 2

Related Work

2.1 Algorithmic Optimizations

Graph algorithms deal with a large amount of data that cannot fit inside a cache of a processor at once. Then again, graph algorithms often necessitate sampling or updating data that is located sparsely and randomly in memory, which dictates usage of slow main memory compared to fast caches. In the case of PageRank, the contributions of a vertex must be spread across vertices that may be located anywhere on memory. In the case of a graph that has low locality, the vast majority of PageRank contributions causes cache misses, which, in turn, causes poor use of memory bandwidth and poor performance. To improve this, there are several solutions, such as the work proposed by Beameret al. [6]. Instead of scattering a vertex's contribution to all edges immediately, they block the ones that are out of reach. Their implementation moves block by block over the graph while the inter-block communications are restricted. They report a performance increase up to 3X by using this approach.

Another notable work that aims to decrease PageRank communication is performed by Lakhotia et al. [7]. They divide graphs into partitions that are stored in memory by an optimized layout. Then, they treat communication between

partitions as messages which are accumulated/delayed. They can decrease the communication volume up to 1.7X while increasing the speed up to 2.7X.

Naive implementations of graph algorithms such as top-down Breadth-First Search (BFS) may visit all edges despite only a small portion of them being valid children. However, it is possible to perform a complete BFS step without visiting many edges by reversing the search direction. Beamer et al. [8] implement this in their Direction-Optimizing Breadth-First Search (DOBFS) work. They developed a heuristic that estimates the steps in which the search direction of BFS should be inverted to visit fewer edges. By doing this, their implementation can visit fewer edges and double the speed.

Naive implementations of the Connected Components (CC) algorithm visit the same vertices multiple times to check if they should be included in a component. There are proposed solutions to this problem that develop more than one component at a time and can move vertices between components. In the work of Slota et al. [9], visited vertices are labeled with colors while BFS is used to search the graph. The combination of parallel graph coloring routine and BFS increases the efficiency of the CC algorithm. They were able to increase the speed of the CC 20X compared to naive serial approaches.

Another similar work is from Sutton et al. [10], in which, Shiloach-Vishkin algorithm [11] is extended with Subgraph Sampling. Their algorithm iterates over subgraphs and makes decisions whether to connect them or not, which aims to process less number of edges. Their implementation offers speedups up to 67X on CPUs and 23X on GPUs.

2.2 High-Level Synthesis Optimizations

As specialized hardware architectures such as GPUs and FPGAs gain market share, especially in the High-Performance Computing (HPC) market, the demand for fast optimized code for HLS increases. HLS increases the productivity

of programmers while making special hardware devices available to a broader audience. There are a few academic works for HLS optimizations that are published recently. The research of Licht et al. comes to the forefront in terms of comprehensiveness which is published in 2018 [12]. They create a framework of critical optimizations of HLS in a hardware-agnostic way, which are grouped as *pipeline-enabling transformations*, *scalability transformations*, and *secondary transformations*.

2.2.1 Pipeline-enabling Transformations

Loop-carried dependencies, which occur when future iterations of a loop are dependent on the results of the previous iterations are one of the factors that prevent pipelining. Transposing the iteration space is one of the proposed solutions to this [12]. Accumulations are often the reason for loop-carried dependencies [13]. Interleaving accumulations, for example, using an array of accumulators to break dependencies is proposed for this problem [12]. Nested loops are costly when it comes to HLS implementations since they often increase initialization interval. Loop flattening/coalescing that merges nested loops and converts them to flat loops are offered as a solution as well [12]. To decrease critical paths of loops, functions are inlined and cheaper data types are used [14]. Decreased critical path lengths enable low initialization intervals, thereby speeding up the execution [13].

2.2.2 Scalability Transformations

Parallel execution is the key to achieve high speeds in HPC environments. Vectorization enables taking advantage of multiple ALUs or SIMD units inside specialized hardware [15]. Optimizations such as tiling, replication, and streaming data flow enable efficient division, routing, and execution of data in parallel [12].

2.2.3 Secondary Transformations

HLS implementations are not immune to costly cache misses. Their performance is tightly dependent on memory bandwidth and latency [16]. Type-demotion is proposed to decrease the used memory space and bandwidth [17]. Memory over-subscription enables memory channels to be fully utilized [12]. Division of computation and memory activity at the kernel level is another optimization that aims to isolate computational units from memory latency [18]. To achieve this, *producer* kernels are used that pipes data to computational units [12].

Chapter 3

Background

3.1 Iterative Graph Algorithms

Iterative graph algorithms traverse the graphs visiting vertices one by one while checking or updating a state. This state may include a label or a score that is given to each vertex. Sometimes edges may have properties such as direction and weight which are read by the algorithm upon visitation. Discovery of neighboring vertices is often needed to determine the vertices that will be visited next.

3.1.1 PageRank (PR)

PageRank (PR) is a graph algorithm that is developed to rank web pages according to importance [19]. The importance of a web page is measured by how many quality links are pointing to it. PR is an iterative algorithm that is implemented using the gather apply scatter (GAS) framework. In each iteration, contributions of incoming links are accumulated to calculate the next PR scores. The mathematical representation of iterations is given in Equation 3.1 where d is damping factor, N is the number of vertices, PR represents PageRank scores, L is the number of outgoing edges of vertices, i.e., the degree of vertices. Complete

matrix form of this equation is given in Equation 3.2 in which edges are represented by an NxN matrix and PageRank scores are represented by a vector with a length of N.

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{3.1}$$

$$\begin{bmatrix} PR(p_1) \\ PR(p_2) \\ . \\ . \\ PR(p_n) \end{bmatrix} = \begin{bmatrix} \frac{1-d}{N} \\ \frac{1-d}{N} \\ . \\ . \\ \frac{1-d}{N} \end{bmatrix} + d \begin{bmatrix} l(p_1,p_1) & l(p_1,p_2) & .. & l(p_1,p_N) \\ l(p_2,p_1) & .. & .. & .. \\ .. & .. & .. & .. \\ .. & .. & l(p_i,p_j) & .. \\ l(p_N,p_1) & .. & .. & l(p_N,p_N) \end{bmatrix} \begin{bmatrix} PR(p_1) \\ PR(p_2) \\ . \\ . \\ PR(p_n) \end{bmatrix} \tag{3.2}$$

3.1.2 Breadth-First Search (BFS)

Breath-First Search (BFS) is a search algorithm that builds a tree by traversing a graph starting from one vertex which would be the root vertex of the tree [20]. BFS prefers building the search tree by one depth at a time, i.e., it does not move to deeper depths before finishing the previous ones. This is the distinctive feature of BFS that separates it from other search algorithms such as Depth-First Search (DFS).

BFS works by storing a frontier of vertices, which are the vertices that would be located in the current depth of the search tree. At every iteration, children of the frontier vertices are checked if they are not discovered yet. Discovered valid children replace the current frontier for the next iteration. Search continues till all the children are visited. The time complexity of BFS can be given as $O(|V|+|E|)$ since all edges and vertices must be visited in the worst case.

3.1.3 Connected Components (CC)

Connected Components (CC) algorithm aims to discover all components of a graph [21]. Component of a graph is defined as an induced subgraph that consists of vertices connected with edges and has no additional connection to the rest of the graph. Search algorithms such as BFS and DFS are used to build these components from starting vertices.

3.2 Intel's Xeon-FPGA Hybrid Platform

Intel proposed a high-performance hybrid architecture that combines their server-class scalable Xeon CPUs, such as E5-2600, with their FPGAs such as Arria 10 inside a single package which is called Heterogeneous Architecture Research Platform (HARP) [22]. There is a fast QPI bridge between the processor and the FPGA. FPGA has access to main memory through the CPU [23]. An overview of this architecture is given in Figure 3.1.

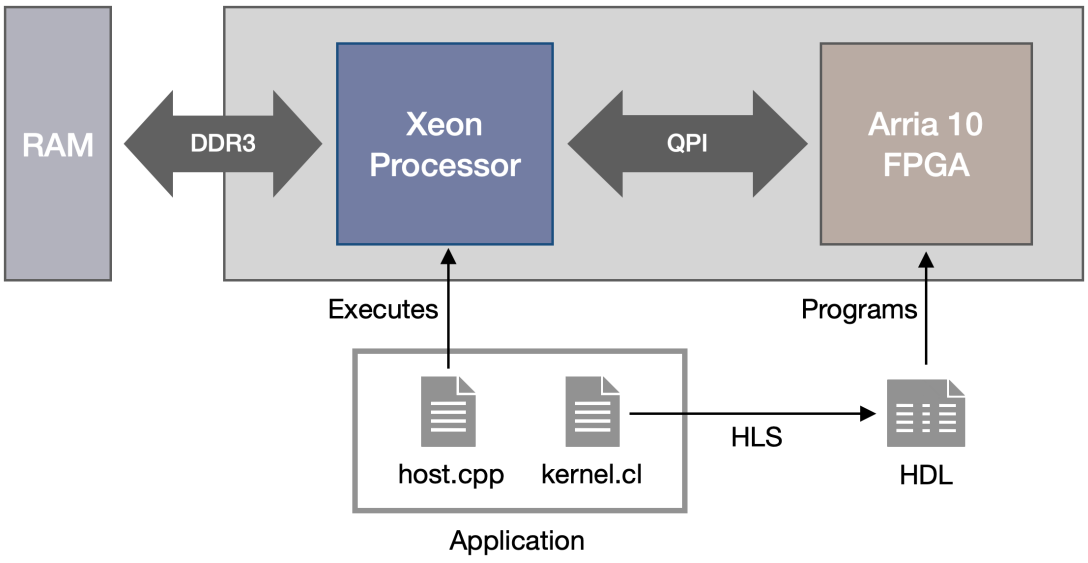

Figure 3.1: Xeon-FPGA hybrid architecture and High Level Synthesis (HLS) workflow.

3.3 High-Level Synthesis (HLS)

High-Level Synthesis is an automated design process that can synthesize digital hardware according to the behavior that can be described by traditional programming languages such as OpenCL [24]. Produced hardware is generally expressed with the Register Transfer Level (RTL) description. It is also possible to obtain Hardware Description Language (HDL) as an output. HLS vastly simplifies the process of programming specialized hardware devices such as FPGAs.

Intel's OpenCL HLS workflow consists of two steps. First, developing a host application that is written in C/C++ that will run on the CPU. Second, developing an OpenCL kernel that will be converted to HDL/RTL by HLS which will be used to program the FPGA device as shown in Figure 3.1. Thus, the programmer would be in charge of only the behavioral and algorithmic parts of the application instead of hardware specifics [14].

3.4 Fixed-Point Arithmetic

Floating-point arithmetic is vital for some of the graph algorithms such as PageRank in which scores are fractional numbers. However floating-point arithmetic may be costly depending on the underlying hardware. Fixed-point arithmetic is developed to circumvent this by using cheap integer arithmetics to emulate fractional numbers [25].

Fixed-point implementations divide digits of an integer into two parts. The significant part of the digits represents the whole part of the fractional number. The less significant part of the digits represents the fraction. The place of division is determined by where *the point is fixed* so to say. This position also determines the scale factor that is used for conversion between floating-point and fixed-point numbers.

Summation and subtraction of fixed-point numbers can be performed with unmodified integer summation and subtraction. Although, many implementations of fixed-points include an overflow handling of some sort [26]. Multiplication and division operations must be implemented in a specialized way which will make use of longer integers that are supported by the underlying hardware. Also, the bits must be shifted according to the scale factor. A simple implementation of fixed-point multiplication that uses a scale factor of 2^{31} is given in 4.5.

3.5 Pipelining

Pipelining is a process that is used by HLS tools to produce digital circuits implemented as RTL/HDL from loop-based programs. HLS creates circuits to perform arithmetic/logic operations inside such loops. Then, it creates pipes which often consist of first-in first-out (FIFO) queues that transport data to/from this circuitry. This process is called pipelining and it is highly susceptible to kernel design which defines the behavior and the logic of the implementation [13].

There is a delay between each iteration of synthesized loops which is called *initialization interval* (II). This delay is affected by the critical path of the circuitry whose length is determined by a combination of arithmetic/logical operations' complexity and specifics of the underlying hardware.

Pipelining is key to achieve good performance in HLS workflows since it enables efficient use of the underlying hardware to the fullest. Factors such as nested loops, loop carried dependencies, etc. prevent good pipelining and impair performance.

Chapter 4

Our Approach

We make use of algorithmic optimizations in an HLS aware fashion while also employing behavioral optimizations to develop fast graph applications. Implementation of the application consists of a host program that runs on CPU and OpenCL kernels that are executed on FPGA.

4.1 High-Level Synthesis (HLS)

Our approach within the scope of the HLS process is given in Figure 4.1. All optimizations are done through the OpenCL kernels, i.e., they are behavioral optimizations, although, they target various levels of the HLS process. The right side of Figure 4.1 depicts which steps of the HLS process are targeted by which optimizations.

The HLS process consists of various steps when combined to generate a bitstream file which is used to program the FPGA. Compiler applies necessary preprocessing to OpenCL codes and perform abstract syntax tree (AST) transformation to obtain an intermediate representation. The effect of algorithmic optimizations enters the HLS process at this point. Moreover, the use of function inlining prevents linker to produce function calls.

Data dependency analysis is an important step since pipelining is constructed here according to the resulting data flow structures. The use of loop flattening simplifies the dependency graph and often results in better pipelining. Choosing the right kernel paradigm and the right direction for the graph algorithms are examples of HLS aware design that takes data flow into account.

The use of multiple kernels enables the allocation/utilization of hardware resources more efficiently. Choice of the data types affects RTL generator's hardware decisions. The clock is affected by all the previous steps. For instance, initialization intervals of loops are selected by the RTL generator while clock frequencies are selected by the bitstream generator. Loops with higher initialization intervals can be clocked faster while consuming more clock cycles per loop iteration.

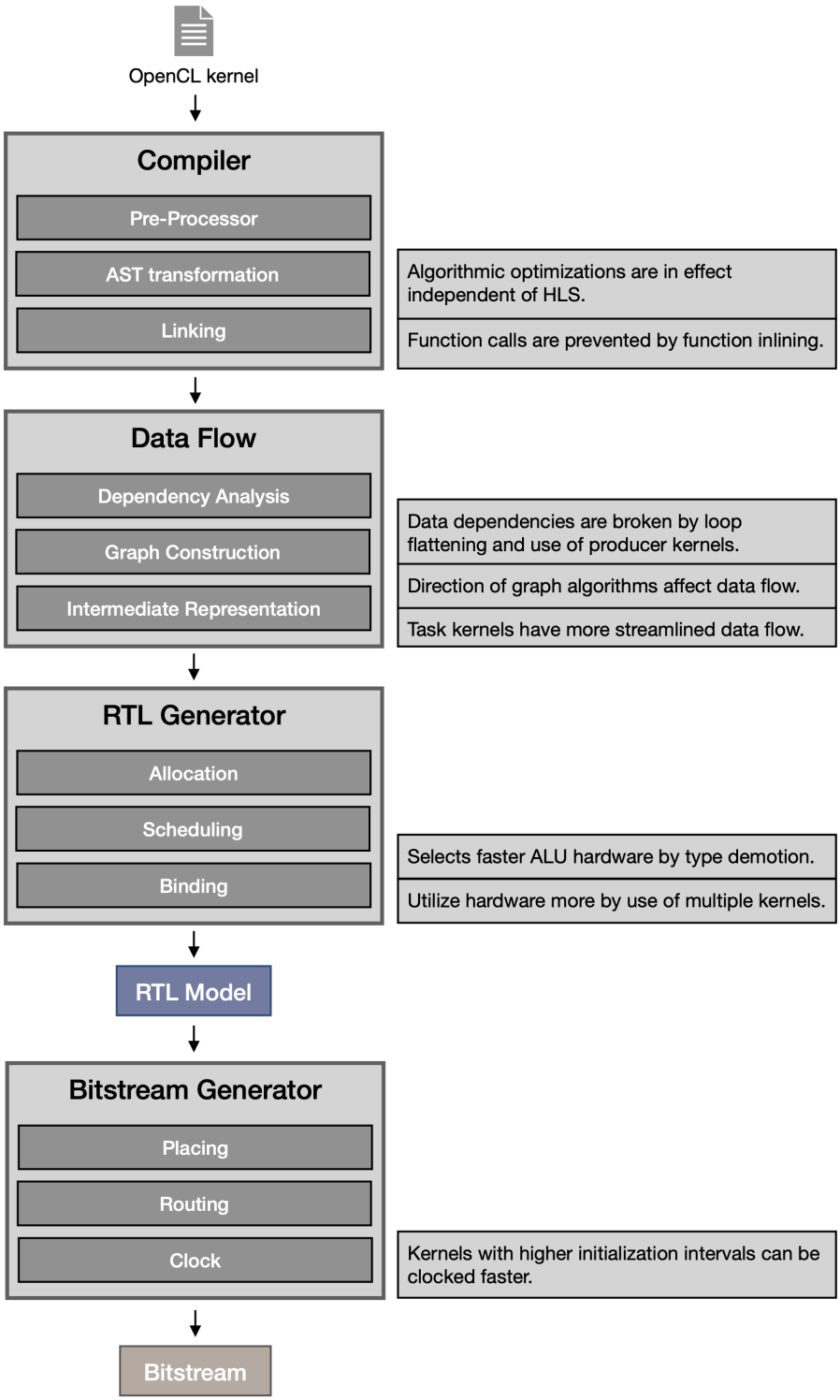

Figure 4.1: Targets of behavioral optimizations in the scope of the HLS process. Our optimization targets are shown on the right side with respect to HLS steps.

4.2 Host Program

Host program is a standard C/C++ application that includes Intel's HLS library's header files which provides access to the required communication framework. It is compiled with Intel's modified GNU compiler, where the FPGA platform is initialized as an OpenCL device. This initialization process includes the creation of kernels using OpenCL files, command and execution queues for the OpenCL kernels, and memory maps that will enable kernels to access the main memory.

Host program performs tasks such as graph generation, memory operations, execution of the baseline algorithms, and gathering and evaluating the results. After graph generation, pre-processing such as the conversion of data types and graph data structures are also done by the host program.

4.2.1 Graph Generation

Graphs are generated or read via The GAP Benchmark Suite (GAPBS) of Beamer et al [27]. GAPBS uses edge list format to store the graphs internally which has a better space efficiency compared to dense graph formats since it requires space only for the existent edges. A directed graph in the edge list format can be traversed in either direction since both the destination and the source vertices of edges are listed. This is an important feature depending on the needs of the graph algorithm. If $|E|$ is the number of edges, the space required for the edge list can be calculated as $2|E|$ indices.

In our implementation, we also utilize the compressed sparse row (CSR) format [28]. CSR format lists only the destination vertices and uses a second list called offsets to map source vertices. Thus, it uses less space which can be calculated as $|V|+|E|$ indices, $|V|$ being the number of vertices. Since $|E|$ is often 8 to 16 times (depending on the degree of the graph) bigger than $|V|$, CSR has a significant space advantage compared to the edge list [29]. But it loses the ability to traverse the graph in both directions. CSR supports a single direction in which the stored

graph can be traversed.

We opted for CSR over the edge list due to its compact structure when traversing the graph in one direction is sufficient. Otherwise, we use the edge list format. Generated graphs are traversed before execution to create the CSR structure from the edge lists.

4.2.2 Compressed Sparse Row (CSR)

In our CSR implementation, graphs are stored in two arrays. The first one is called **edges** which has the size of $|E|$ (number of edges) and lists all destination vertices. The second one is called **offsets** which has a size $|V|$ (number of vertices) and stores an index per each source vertex that locates neighboring edges in **edges**. CSR is said to be in the pull direction in this arrangement since destination vertices are listed in **edges**. If destination and source vertices are switched their places, then CSR is considered to be in the push direction. Thus, CSR has a preferred direction, unlike the edge list format. We use CSR for PR and BFS algorithms, whereas the edge list format is used for Direction Optimizing Breadth-First Search (DOBFS) since a bi-directional traversal is required.

Given vertex i, neighboring vertices of i (incoming edges if CSR in pull direction, outgoing edges if CSR is in push direction) are stored in the range of edges[offsets[i-1]] - edges[offsets[i]] as shown in Figure 4.2.

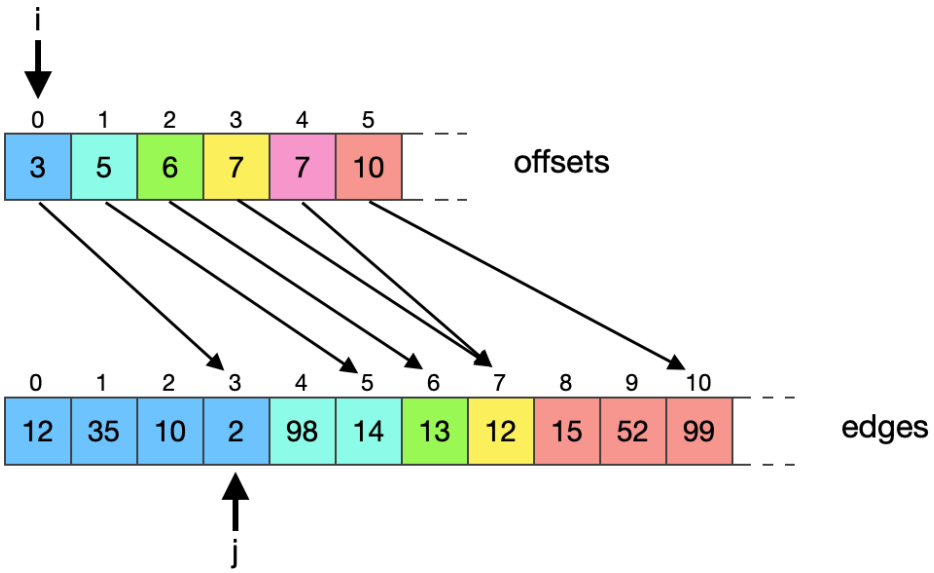

Figure 4.2: CSR format stores the graph in two arrays compactly.

4.3 OpenCL Kernels

OpenCL kernels are program pieces or functions that define a unit of work that is targeted to be executed on an OpenCL device. These kernels are written in OpenCL which has a syntax similar to C language with additional pragmas.

4.3.1 Kernel Paradigm

HLS workflow supports two types of programming paradigms when it comes to kernels, namely, NDRange and task. In the case of NDRange, a kernel is responsible for one piece of data such as a vertex in a graph, similar to CUDA [14]. In this scenario, multiple kernels are being executed and each kernel handles a small part of the data. In the case of task kernels, a kernel has access to all of the data and runs iteratively over it. Intel suggests using task kernels for FPGAs [13]. Thus, our implementation uses task kernels exclusively.

4.3.2 Multiple Kernels

Multiple task kernels can be run in parallel and controlled independently from each other. Considering the fact that the speed of a kernel is limited by the data transfer rates and calculation delays, running multiple instances of the same kernel is logical. If the number of kernels is sufficient, memory bandwidth can be fully utilized through memory over-subscription.

Graph algorithms that traverse the graph, such as in BFS, or iterate over vertices, such as in PR, can be divided amongst multiple task kernels that are running in parallel as shown in Listing 4.1. In our implementation, we experimented with a various number of kernels. The range of indices that each kernel would be responsible for can be calculated by the host program and passed to kernels respectively.

```
1  //single task kernel
2  for (int i = 0; i < last_vertex; i++){
3     vertices[i] = ....
4  }
5
6  //multiple task kernels
7  //kernel 0
8  for (int i = start_vertex[0]; i < last_vertex[0]; i++){
9     vertices[i] = ....
10 }
11 .
12 .
13 //kernel k
14 for (int i = start_vertex[k]; i < last_vertex[k]; i++){
15    vertices[i] = ....
16 }
```

Listing 4.1: Multiple kernels running simultaneously.

4.3.3 Producer Kernels

HLS automatically handles memory reads and writes which include automation of queuing load/store requests, caching, and pipelining [14]. Even so, explicitly programming memory requests can help with the pipelining of the kernels. For both PR and BFS, vertices are iterated sequentially which can be easily pipelined but neighboring edges need to be read randomly which may cause pipelining to fail. To avoid this, two kernels are programmed: one is the producer kernel which reads edges and pipes them to the other kernel. The other kernel reads edges from the pipe and operates on them as shown in Listing 4.2. The synthesis of this code creates a FIFO queue between two kernels that is sufficiently long to account for a random edge read latency.

```
1  //Producer Kernel
2  for(int i = 0; i < last_vertex; i++){
3     firstEdge = offsets[i]
4     lastEdge = offsets[i+1]
5     write_channel(firstEdge);
6     write_channel(lastEdge)
7  }
8  //Executing Kernel
9  firstEdge = read_channel()
10 lastEdge = read_channel()
11 for(int e = firstEdge; e < lastEdge; e++){
12    pagerank[i] += pagerank[e]
13 }
```

Listing 4.2: Producer kernel pipes data to executing kernel.

4.3.4 Loop Flattening

Nested loops sometimes cannot be pipelined by HLS or they can be pipelined with the cost of longer initialization intervals [13]. Listing 4.3 depicts such a PR implementation that cannot be properly pipelined with HLS. To avoid this, graphs must be traversed by flat loops. This is done by iterating over **offsets** and **edges** arrays at the same time using two indices, i and j, as shown in Listing 4.4] and illustrated in Figure 4.2.

```
for(int i=0; i < last_vertex; i++){ //iterate through vertices
   first_edge = offsets[i-1]
   last_edge = offsets[i]
   //iterate over incoming edges per vertex
   for(int j = firstEdge; j <= last_edge; j++){
      PR[i] += PR[j]
   }
}
```

Listing 4.3: PageRank using nested loops.

```
int i = 0 //first vertex
int j = 0 //first edge
while(j < last_edge){ //run over all edges
   PR[i] += PR[e]
   j++ //go to next edge

   if(j>offset[i]){
      i++ //move to next vertex
      PR[i] = PR[i]/degree[i]
   }
}
```

Listing 4.4: PageRank implemented with a single flattened loop.

4.3.5 Type Demotion

PageRank algorithm deals with fractional numbers. Because of this, storing PageRank values in floating-point format is sensible. Since the PageRank algorithm requires summation and multiplication of floating-point numbers, the cost of these arithmetic operations in underlying hardware is crucial to obtain a fast implementation.

Similar to CPUs, floating-point operations are costlier compared to integer operations in FPGAs. Since arithmetic operations are part of the kernel pipeline, floating-point operations add latency to iterations. Fixed-point libraries aim to solve this problem by providing low precision fractional numbers which are implemented by cheap integer operations.

We developed two different implementations of the PageRank algorithm that use fix16 and float32 numbers to store PageRank values. The *libfixmath* library is used in fix16 implementation [26].

4.3.6 Function Inlining

Function calls that are made inside kernel loops may cause call overheads that may, in turn, decrease the speed of the execution. To prevent this, functions must be inlined in which case, compiled code of the functions are placed directly inside the loops and no function calls occur.

```
1 inline uint mul(uint u1, uint u2)
2 {
3   return (uint) ((((ulong)u1)*((ulong)u2)) >> 31);
4 }
5 //inside loop
6 acc += mul(PR, degree)
```

Listing 4.5: Function inlining in our implementation.

Chapter 5

Implementations

5.1 PageRank (PR)

5.1.1 Direction

The PR algorithm is implemented in both pull and push directions as shown in Listing 5.1 and 5.2. In the pull direction, incoming contributions are accumulated and the PageRank scores are updated once; in the push direction, the contribution of each vertex is pushed multiple times to all outgoing edges, i.e., scores are updated multiple times.

Resulted pipelines that are synthesized by HLS are shown in Figures 5.2 and 5.1. There is a significant memory latency in the case of PageRank score reads and writes. To account for this, HLS creates a FIFO queue between memory and kernel whose role and location in the pipeline are dictated by the direction of the PR algorithm.

```
for(int i=first_vertex; i < last_vertex; i++){
   first_edge = offsets[i]
   last_edge = offsets[i+1]
   //iterate over outgoing edges per vertex
   for(int e = firstEdge; e < last_edge; e++){
      //push current vertex's contribution to edges
      pagerank[e] += pagerank[i]
   }
}
```

Listing 5.1: Algorithm in the pull direction.

```
for(int i=first_vertex; i < last_vertex; i++){
   first_edge = offsets[i]
   last_edge = offsets[i+1]
    //iterate over incoming edges per vertex
   for(int e = firstEdge; e < last_edge; e++){
      //pull PageRank contribution of edges
      pagerank[i] += pagerank[e]
   }
}
```

Listing 5.2: Algorithm in the push direction.

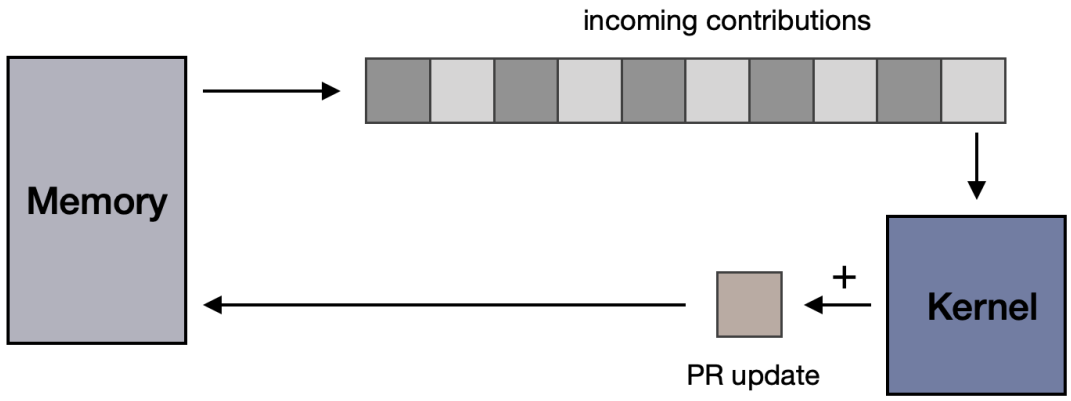

Figure 5.1: PageRank implementation in pull direction.

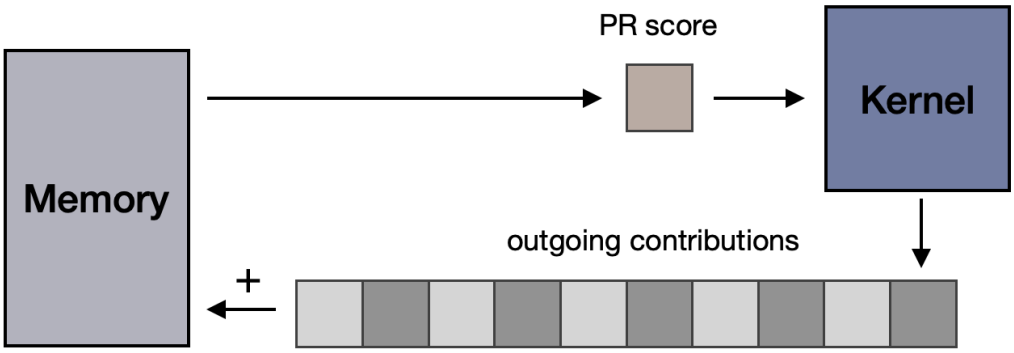

Figure 5.2: PageRank implementation in push direction.

5.1.2 Storing PageRank/Degree instead of PageRank

To calculate the contribution of a vertex, its PageRank score has to be divided by its degree as shown in Listing 5.3. Thus, a degree value must be read in addition to the score for each edge. Our implementation avoids this by storing PR/degree values as PageRank scores as shown in Listing 5.4 and minimizes the amount of data read from memory.

```
1 for(e:incoming(v))
2    PR[i] += PR[e]/degree[e]
3
```

Listing 5.3: Storing PageRank scores.

```
1 for(e:incoming(v))
2    PR[i] += PR[e]
3 PR[i] = PR[i]/degree[i]
```

Listing 5.4: Storing scores as PageRank/Degree.

5.1.3 Handling Special Cases

There are some special cases in the PR algorithm such as when a vertex has no incoming edges hence no contributions. When this case transpires, the kernel should give a base PageRank score to the vertex and move on to the next vertex. Also, note that, this branch can be calculated beforehand to make pipelining easy by a producer kernel.

```
1
2 if ( is_zero_in ){ //there is no incoming edge for the current vertex
3    pagerank[i] = c //a constant value
4    i++ // move to next vertex
5 }
6 else{
7    pagerank[i] += pagerank[e]
8    e++ //keep iterating through incoming edges of current vertex
9 }
```

Listing 5.5: Special cases handled by our approach.

5.2 Breadth-First Search (BFS)

5.2.1 Implementing the Frontier Queue

Implementation of the BFS algorithm necessitates the usage of a queue that stores newly discovered vertices which are called frontier. The size of this queue reaches its maximum generally towards in the middle of the BFS execution. The maximum allowed size of the queue must be big enough to accommodate this widest frontier. Implementing such a queue for HLS can be a very complex task considering pipelining. Thus, instead of the queue, our implementation uses an array to mark if a vertex is in the frontier. This array stores an integer per vertex that indicates the location of the vertex in the BFS tree. We call this value the

level. Level 0 means that vertex is not discovered, whereas level N means that vertex is discovered at the level N and if the current level of the BFS is equal to N then vertex is said to be in the frontier. This array of levels is initialized to zero except for the root vertex, which is initialized to 1 to mark it as the starting vertex.

The implemented BFS algorithm iterates through all vertices and checks if any of them should be in the frontier. If a vertex is in the frontier, outgoing edges are fetched and neighboring vertices are checked if they are discovered previously. Undiscovered vertices are marked as the frontier for the next iteration of the algorithm. This way, potentially complicated enqueue/dequeue operations are omitted. Instead, frontier, and discovered/undiscovered vertices are stored in one simple array which can be easily pipelined and synthesized.

Figure 5.3 depicts how the array of levels is changing throughout BFS execution steps. In the first step, all levels are equal to 0 which is marking the vertices as undiscovered. The level of the root vertex is marked as 1 which indicates that it is in the frontier of BFS's first step. The search begins with the root, vertex 5. Adjacent vertices 2 and 8 are discovered and marked for the next frontier. Then BFS moves to the second step and repeats the search. The current level indicates the current search depth of the BFS algorithm. Vertices that will be in the next frontier are marked with the level of *current level+1*. When BFS moves to the next step, the current level is increased by one. At each level, vertices marked with that level are in the frontier. Therefore, frontier, undiscovered vertices, and levels in which discovered vertices are located in the BFS tree are stored in one array. Moreover, iterating through an array instead of dealing with queues simplifies the HLS process.

Some BFS implementations are using bitmaps to store if vertices are discovered or not because they are cheaper than using bytes or integers [30]. But memory latency is the main factor that dictates the speed in the case of graph algorithms as opposed to memory bandwidth and space [8]. Our implementation does not utilize bitmaps in favor of the array of levels.

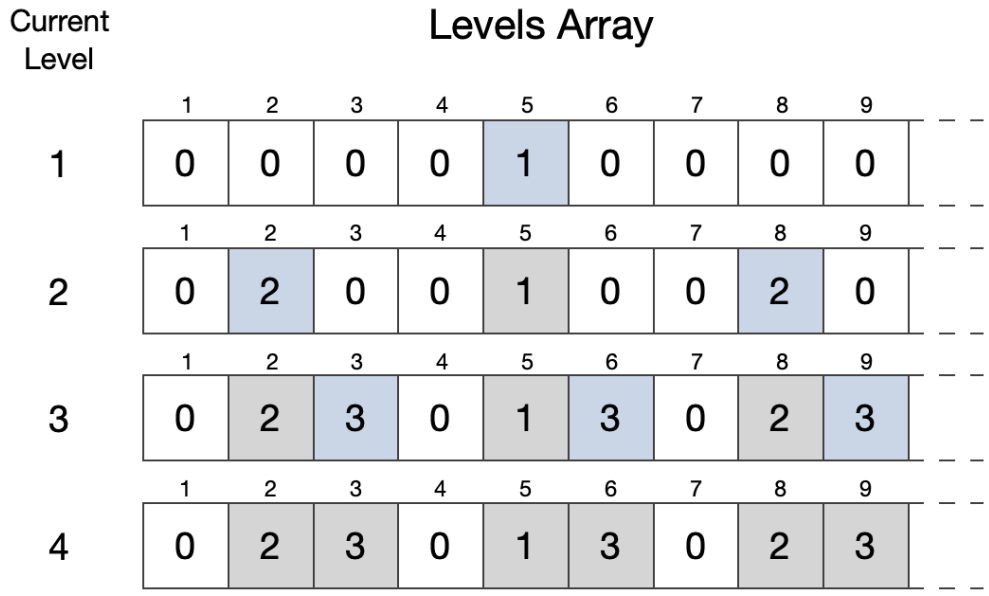

Figure 5.3: The array of the levels throughout the BFS execution steps.

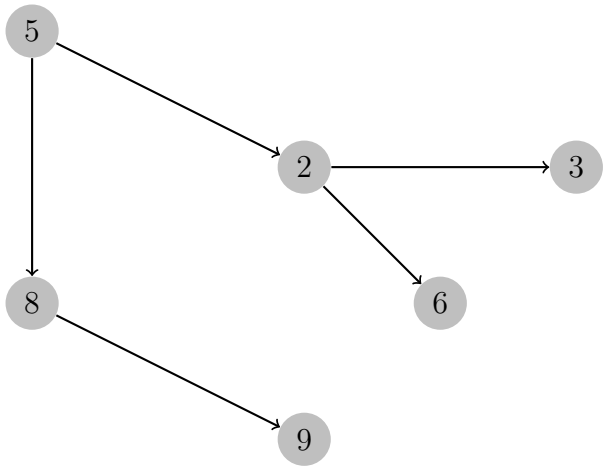

Figure 5.4: The search tree of the BFS.

```
int current_level = 1
while(...){ //continue until no new discoveries
   for (int i = 0; i < last_vertex; i++ ) { //iterate through vertices
      if( levels[i] == current_level ){ //check if in the frontier
         for(int j = offsets[i]; j < offsets[i+1]; j++){
            if( levels[ edges[j] ] == 0){ //check if undiscovered
               levels[ edges[j] ] = current_level+1 //enqueue
            }
         }
      }
   }
   current_level++
}
```

Listing 5.6: BFS implementation

5.2.2 Direction-Optimizing Breadth-First Search (DOBFS)

After the BFS algorithm's first couple of steps, frontier size increases to a size that is comparable to the number of undiscovered vertices. In such a situation, it is better to invert the direction of the BFS algorithm, i.e., instead of finding undiscovered children of the frontier, it can be checked whether the parents of the undiscovered vertices are in the frontier. If the number of undiscovered vertices is less than the size of the frontier, then the BFS algorithm will check less number of vertices, thus run faster [8].

To enable the inversion of the direction of the search, two implementations of a BFS step are developed. One is called Top-Down (TD) which is the usual direction in which BFS is run that checks child vertices of the frontier. The other one is called Bottom-Up (BU) which is the reverse direction that parents of undiscovered vertices are checked. The main BFS loop keeps track of the current size of the frontier and the size of undiscovered vertices. At the beginning of

each BFS step, a decision is made regarding whether to employ the TD or BU according to the size comparison of the frontier and undiscovered vertices.

In this arrangement, the algorithm would employ TD in the first couple of steps since the frontier will be small at the beginning. After the frontier gets larger, it will switch the direction to BU. When frontier gets small in size towards last steps, TD is employed again.

In both TD and BU approaches, the source vertex of an edge is checked if it is in the frontier and the destination vertex is checked if it is not yet discovered. TD branches first with the frontier check since it goes from parent to child. BU branches first with a discovered check since it goes from child to parent. So, TD and BU employs the same two branches but in reverse order.

The structure of the CSR format does not allow traversing a graph in both directions, thus, TD-BU implementation of the BFS algorithm stores the graphs in edge list format.

Also, note that, we omit to use any type of queue for bitmap conversion which can be found in Beamer's BFS work [8] since we employ an array of levels instead of a queue which enables the selection of both frontier and undiscovered vertices in an iterative manner that can be easily pipelined [9].

```
int current_level = 1
int frontier_size = 1
int not_discovered_size = V-1
while( frontier_size != 0){ //continue until no new discoveries
   if ( frontier_size < not_discovered_size ){ //Top Down
      frontier_size = 0
      for (int i = 0; i < last_vertex; i++ ) {
         if( levels[i] == current_level ){ //check if in the frontier
            for( v : outgoing(i) ){
               if( levels[v] == 0){ //check if not discovered previously
                  levels[v] = current_level+1 //enqueue
                  frontier_size++
                  not_discovered_size--
               }
            }
         }
   }}
   else{ // Bottom Up
      frontier_size = 0
      for (int i = 0; i < last_vertex; i++ ) {
         if( levels[i] == 0 ){ //check if not discovered previously
            for(v : incoming(i)){
               if( levels[v] == current_level){ //check if in frontier
                  levels[i] = current_level+1 //enqueue
                  frontier_size++
                  not_discovered_size--
               }
            }
         }
   }}
   current_level++
}
```

Listing 5.7: Direction-Optimizing Breadth First Search (DOBFS).

5.3 Connected Components (CC)

The Connected Components implementation relies either on BFS or on Direction Optimizing BFS (DOBFS) which are used to develop components from a starting vertex. Components are developed sequentially. After a component is developed, another one is started with a root vertex that is randomly chosen from undiscovered vertices. Multiple kernels are run simultaneously when developing a component to obtain high speeds. Simplified flow of CC implementation is given in Listing 5.8.

```
1 component_id = 1
2 while( 0 < undiscovered_size){
3    root_node = choose(undiscovered_vertices())
4    DOBFS(root_node, component_id)
5    component_id++
6 } //continue this loop until all vertices are discovered
```

Listing 5.8: Implementation of Connected Components.

Chapter 6

Experimental Evaluation

6.1 Setup

6.1.1 Hardware

We carried out our experiments on a heterogeneous platform which combines an Intel Xeon multicore CPU at 2.4GHz with an Arria 10 FPGA in one package. This test setup runs a Linux operating system with Intel drivers and has a 36GB of RAM [31].

6.1.2 Graphs

Graphs used in our experiments are listed in Table 6.1. We preferred to use big, random, and uniform graphs to test our implementations. We have 4 synthetic graphs that have the size of $|V| = 2^{25}$ and they come in the combinations of being directed/undirected and having degrees of 8.0/16.0. We also have 2 real-world graphs. Our implementations can be evaluated comprehensively thanks to the diversity of these graphs.

Facebook and *Wikipedia* graphs are used to give a sense of the real-world performance of the graph applications. These are relatively small, not uniform, have average degrees of 9.2 and 22.8, respectively. *Wikipedia* is directed since its edges represent hyperlinks between pages, while *Facebook* is undirected since it is a social network graph. These graphs are built with the help of web-crawlers and available in various graph databases to download [32, 33].

erdos25 is a graph generated using the uniform-random model of Erdos Reyni [34]. *gapbs25* is the directed version of this graph, which is produced via a small modification made to GAP Benchmark Suite [27]. *kron25* is generated using Kronecker synthetic graph generator using Graph 500's default parameters [35, 36]. *rmat25* is generated using the recursive matrix work of Chakrabarti et al. [37]. GAP Benchmark Suite is used as graph loading software to import or generate all of the graphs used in this evaluation. [27].

Table 6.1: Properties of the graphs used for evaluation.

Abbreviation	Graph	Vertices (M)	Edges (M)	Degree	Directed	Ref.
kron25	Kronecker	33.554	536.870	16.0	N	[36, 35]
erdos25	Erdos–Reyni	33.554	268.435	8.0	N	[38, 39]
rmat25	RMAT	33.554	268.435	8.0	Y	[38, 37]
gapbs25	GAPBS	33.554	536.871	16.0	Y	[27]
facebook	Facebook Trace A	3.097	28.377	9.2	N	[32]
wikipedia	Wikipedia Links	5.717	130.160	22.8	Y	[40, 33]

6.1.3 Measurements

We need a generic way of indicating the performance of different types of graph algorithms so that speedups can be compared. Because of this, we cannot use performance indicators like the duration of completion. Instead, we measure the executed/traversed number of edges during certain parts of the algorithms. To achieve this, steps of the graph algorithms are timed such as one pass over all vertices in PR or one pass over all frontier vertices in BFS. The number of edges

iterated over in steps is divided by these durations. Then values of all the steps are averaged to get a single performance figure in terms of edges/second.

We use GAP Benchmark Suite's PR, BFS, and CC implementations as our baseline which are executed in the host program on the CPU.

6.2 Experimental Results

We only show performance figures of kernels that are pipelined with reasonable initialization intervals, specifically, in the range of 1 to 4. A kernel that failed to be pipelined properly has an initiation interval of a couple of hundred cycles which in turn means that it will run a couple of hundred times slower. We use standardized benchmarks as our baseline instead of using these too slow implementations.

To see the effect of memory over-subscription on performance, graph algorithms are tried with various numbers of kernels as shown in Figure 6.1. For all of the algorithms, there are significant performance increases when the number of kernels are increased up to 4. On the other hand, performance converges when the number of kernels is changed from 4 to 8. This indicates that 4 kernels are enough to utilize the memory bandwidth. We observe diminishing returns when using more than 4 kernels despite there is a slight performance increase when moving from n=4 to n=8.

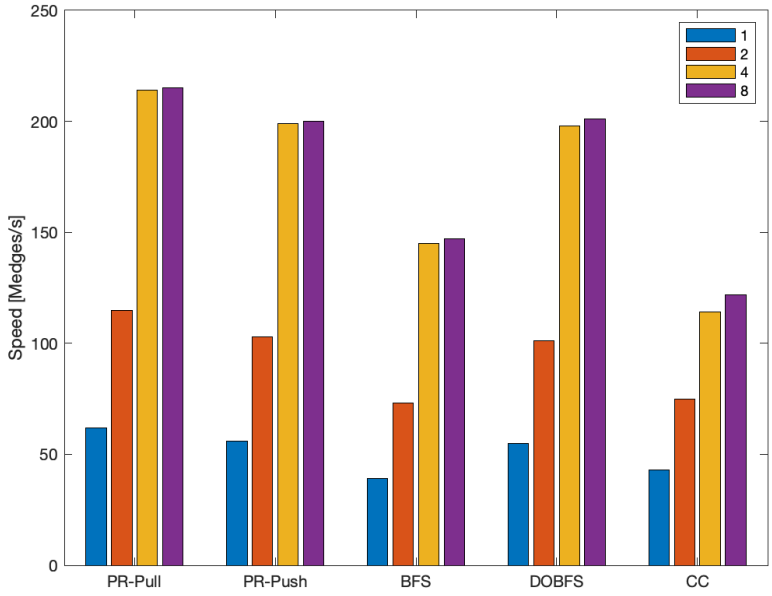

Figure 6.1: The effect of number of kernels on speed.

We showed how to implement PR and BFS algorithms in both nested and flat loops in the previous section. We also demonstrated how to create producer kernels to order memory requests explicitly.

In our experiments, we observed that the initiation interval (II) of nested loops is ranging from 2 clock cycles to 4 clock cycles while the flat loop manages to achieve an initialization interval of 1 clock cycle. We observed up to 2X speedups in case of the flat loops compared to nested loops as shown in Figure 6.2.

We also tried the producer kernel versions of both nested and flat loop implementations. Once a kernel is pipelined well with the minimal initialization interval, producer kernels do not create significant differences in speed as shown in Figure 6.2. Also note that, when producer kernels are nested, we continue to pay the price of long initialization intervals. Thus, creating producer kernels is not a solution to sub-optimal pipelining. Nevertheless, when HLS is unable to pipeline a kernel with minimal initialization intervals due to some conflict, producer kernels can help HLS to match the programmer intention better.

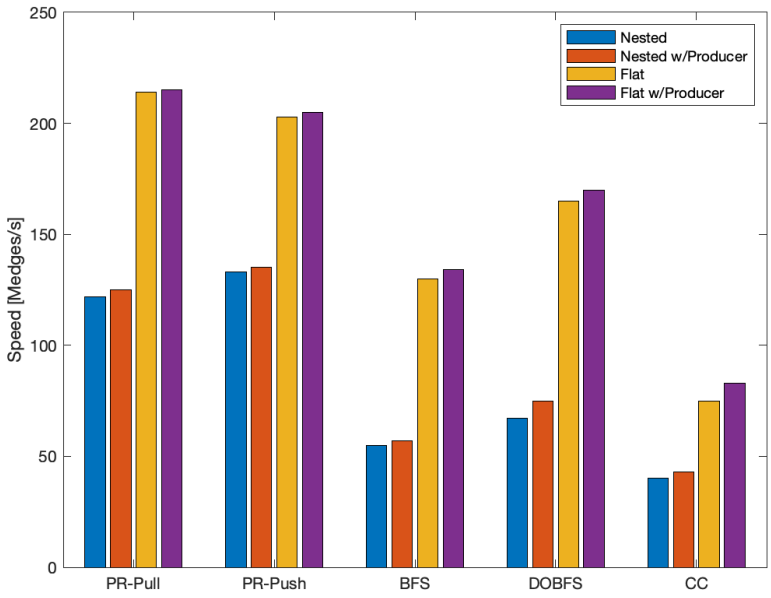

Figure 6.2: The effect of loop structure on speed.

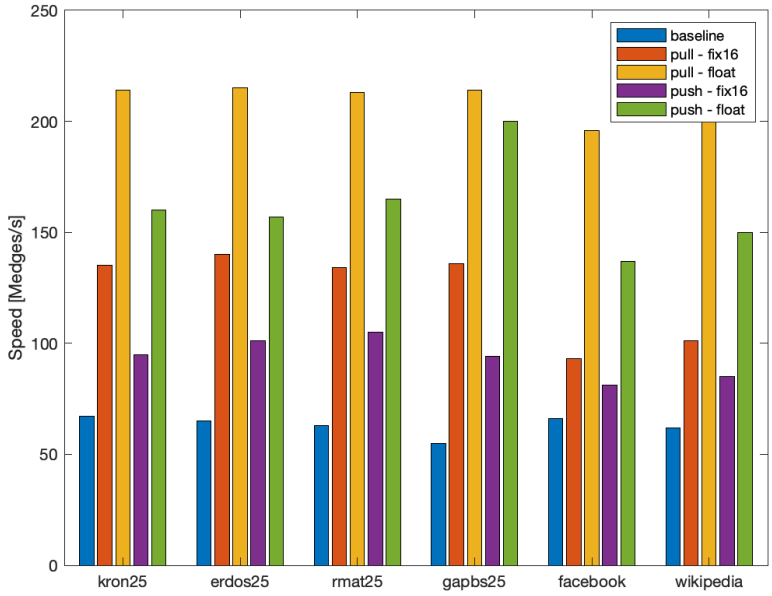

Figure 6.3: Comparison of different PR implementations.

Figure 6.3 depicts the performance of several PR implementations which are selected to be the best versions in terms of pipelining and memory bandwidth utilization. PR in pull direction is consistently faster than push direction. In both directions, $|E|+|V|$ sequential memory reads are performed to traverse the graph. To calculate and update PR scores, some additional random score reads and writes must be performed which are listed in Table 6.2. Since more random memory activity must be performed in the pull direction, it is expected that PR runs faster in the pull direction.

Table 6.2: The number of the memory operations needed for PR score updates.

Task	Pull	Push
Traverse Graph	$(\|E\|+\|V\|)$ reads sequential.	$(\|E\|+\|V\|)$ reads sequential
Get Contributions	$\|E\|$ reads random	$\|V\|$ reads sequential
Update Scores	$\|V\|$ writes sequential	$\|E\|$ (reads+writes) random

Comparing performances of fix16 and floating-point PR implementations, we encountered an unexpected result. Specifically, the floating-point implementation of PR was significantly faster than fix16 despite fix16 operations are notably cheaper than floating-point operations. Moreover, the initialization interval of synthesis of floating-point implementation was 4 cycles compared to fix16's only 1 cycle. Thus, we expect fix16 to be faster up to 4 times.

This becomes more clear when kernel frequencies are considered. HLS is in charge of kernel frequencies and it determines them according to lengths of critical paths. HLS increases the frequency of floating-point kernel since its critical path is divided into 4 clock cycles, i.e., costlier floating-point operations can be performed in 4 clock cycles duration while each clock cycle can be faster.

In our FPGA and HLS setup, one clock is shared between all kernels and also memory controllers. Thus, increasing kernel frequency increases memory speed and all other operations. To observe this effect more closely, PR speeds and kernel frequencies are plotted respectively in Figure 6.4 which demonstrates that there is a clear relationship between the PR performance and kernel frequencies.

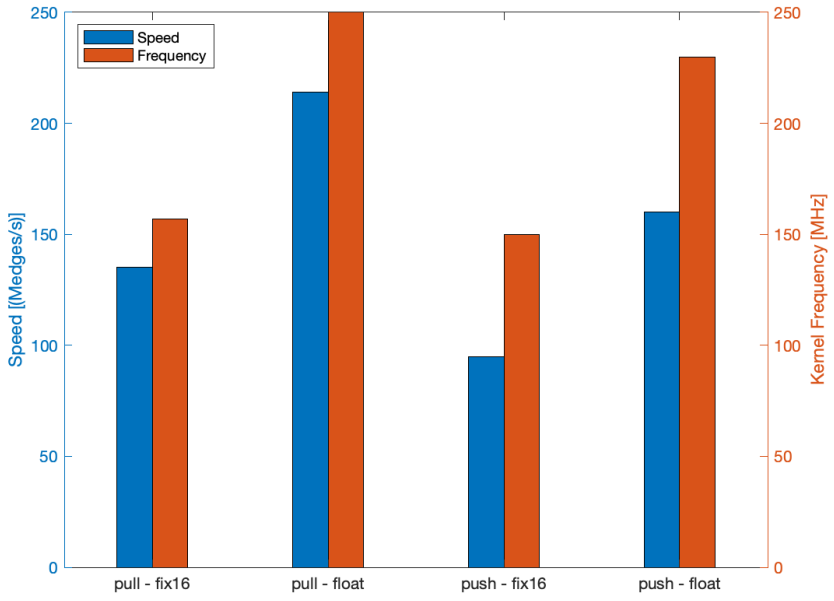

Figure 6.4: Comparison of speeds and frequencies of PR kernels.

Figure 6.5 illustrates the performance comparison of BFS, DOBFS, and the BFS implementation of GAP Benchmark Suite which is used as baseline [27]. BFS is not so much of an improvement over the baseline while DOBFS can enable substantial speedups since it switches its BFS direction when the frontier becomes larger than the number of undiscovered vertices, where the bottom-up approach is preferred instead of top-down, i.e., the algorithm tries to find parents of undiscovered vertices instead of finding undiscovered children of frontier vertices. This decreases number of iterations required for big frontiers in the middle steps of BFS, thereby providing significant speedups.

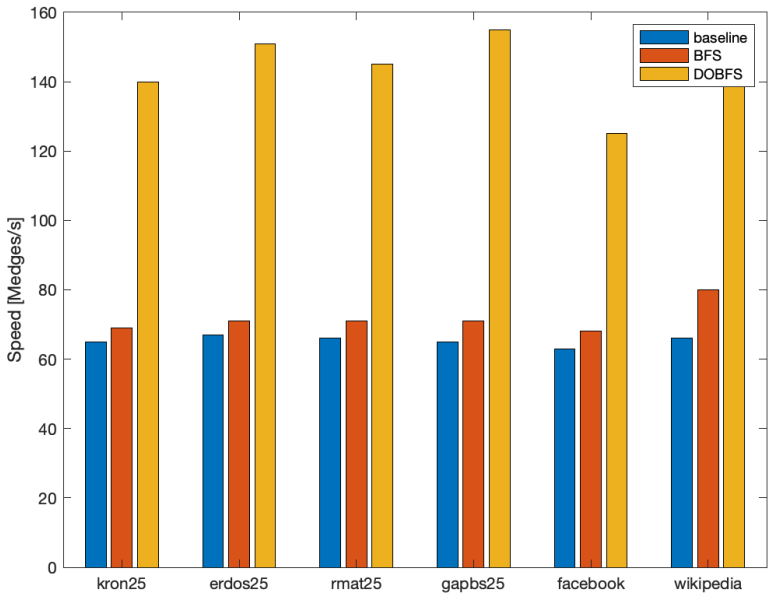

Figure 6.5: The performance of different BFS implementations.

This behavior is demonstrated in Figure 6.6 which plots the size of the frontier and the number of undiscovered vertices throughout the BFS steps for the gapbs25 graph. The frontier is generally small except in the middle steps when the BFS algorithm moves to deeper levels. The size of the undiscovered vertices decreases throughout the BFS steps, although, faster when the frontier is big since the frontier vertices are discovered in each step. Moreover, the plot is shaded to indicate the preferableness of the top-down and the bottom-up approaches according to the difference between the size of the frontier and the undiscovered vertices. Frontier is small in the red regions, thus, the top-down approach is faster than the bottom-up one. On the other hand, in the green regions, the bottom-up is preferable since the frontier is too big.

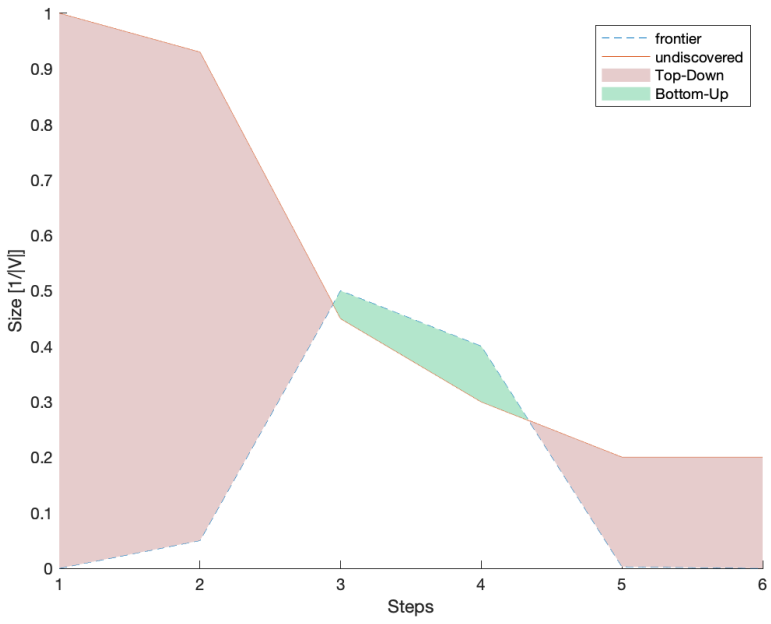

Figure 6.6: The frontier size of the Breadth-First Search (BFS) during different execution steps.

The performance comparison of different implementations of CC is given in Figure 6.7. These results coincide with BFS and DOBFS since they are the backbone of CC implementation. Also, note that, the CC algorithm has additional tasks such as selecting the start vertices and building components that makes it significantly slower than BFS implementations.

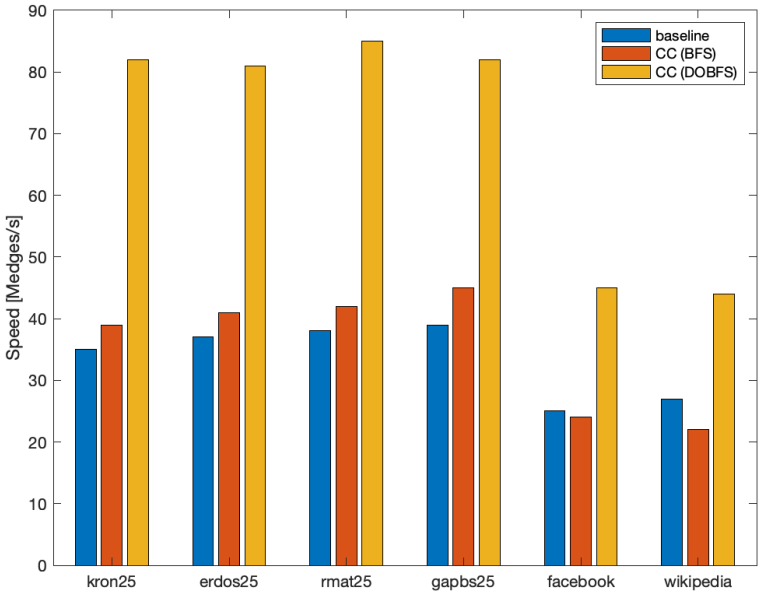

Figure 6.7: The performance comparison of different CC implementations.

6.2.1 Simultaneous execution of CPU and FPGA

We executed our algorithms on the CPU and the FPGA at the same time to see if we can reach a total performance that outperforms FPGA-only scenario. We compared CPU and FPGA speeds when they are running alone and when they are running at the same time as shown in Figure 6.8.

When the two resources are used simultaneously, CPU performance decreases slightly while the FPGA performance falls dramatically to several Medges/s. This mainly stems from the fact that the CPU consumes most of the memory bandwidth. During our tests, we also noticed that when the CPU finishes execution, FPGA immediately gains significant speed. In light of these results, we conclude that the best performance is achievable by running the FPGA alone.

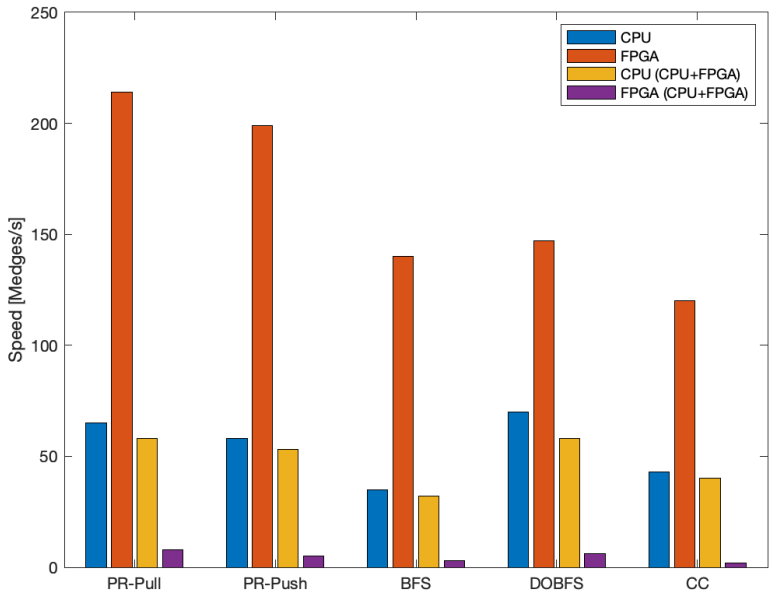

Figure 6.8: Performance comparison of CPU-only, FPGA-only, CPU+FPGA execution scenarios.

To further analyze this, we run CPU+FPGA execution scenario with various numbers of kernels to observe the effect of memory bandwidth on performance interplay between CPU and FPGA. Expectedly, total performance does not change while FPGA does more of the work as the number of kernels increases as shown in Figure 6.9

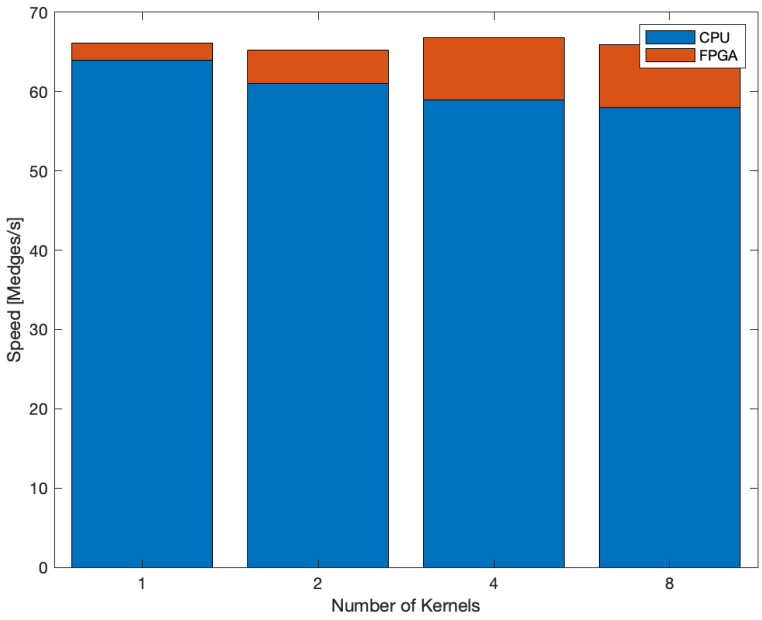

Figure 6.9: Performance of CPU+FPGA with different number of kernels.

6.2.2 Discussion

Note that, time spent in memory transfers is also included in our measurements. Therefore, some of the overheads dependent on the graph size. To measure this, we tried our kernels with graphs with sizes ranging between 2^{10} to 2^{25} vertices as shown in Figure 6.10. Performance of the algorithms increases as graphs get bigger since the associated communication overheads get relatively smaller. For instance, the PR algorithm spends a significant amount of time between steps as new scores need to be copied. Also, the total score difference needs to be calculated to decide if the end of the PR is reached. Kernels are paused during these tasks. With bigger graphs, FPGA spends more time on kernel tasks compared to these off-kernel tasks.

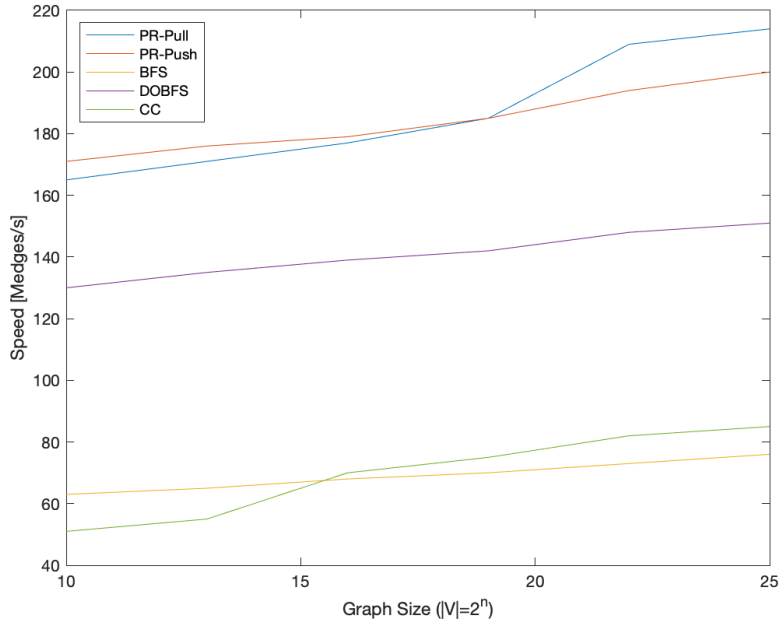

Figure 6.10: Performance change with various graph sizes.

We used CSR format for PR, BFS, and CC while using the edge list format for DOBFS and CC (DOBFS). We compare memory footprints of these graph structures to give an idea about the memory requirements of graph algorithms in Figure 6.11. The edge list format requires more memory since it contains both source and destination vertices, while CSR contains only the destination vertices. As mentioned before, CSR has a preferred direction which is a disadvantage while the edge list can be traversed in both directions.

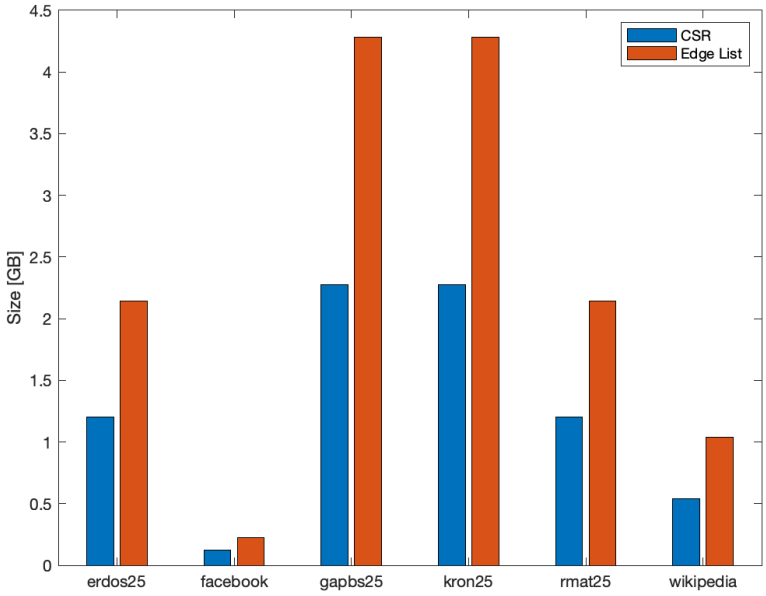

Figure 6.11: Memory footprints of CSR and edge list formats.

Chapter 7

Discussions and Future Work

The experimental results demonstrate that performant implementations of PR, BFS, and CC that run on an FPGA can be developed with OpenCL using the HLS workflow. Developing well-pipelined kernels combined with algorithmic optimizations is a key to this.

We observed that if the number of kernels increased beyond 4, performance does not continue to increase. Considering the fact that it is possible to fit more kernels into the FPGA, we are not able to fully utilize the available hardware. This is caused by the limited memory bandwidth and high latencies. To increase performance further, the memory usage of the algorithms must be optimized, i.e., memory connectivity must be decreased algorithmically. As future work, memory bandwidths and latencies of the CPU and the FPGA must be cross-examined to achieve further improvements.

Moreover, we executed the applications simultaneously on the CPU and the FPGA to obtain a performance increase. However, we did not see a real improvement over our baseline implementation. This follows from the fact that, the FPGA already consumes the available memory bandwidth alone. On a different hardware, say a CPU+FPGA hybrid, in which memory is not shared, instead FPGA has its own memory, this experiment is expected to yield performance

improvement associated with running CPUs and FPGAs at the same time.

In the case of PR, the best performance is achieved by using *flat loop*, *floating-point numbers*, *4 kernels*, *pull-based implementation*. It is reasonable to expect the best results using this configuration, except that the floating-point being faster than the fixed-point was unexpected. However, as explained, this is primarily due to the kernel frequencies. We observed that the memory speed of the FPGA is highly dependent on the kernel frequency which affects the overall performance dramatically.

We realized that HLS uses a common clock for both kernels and memory controllers. In such a situation, the programmer's aim should be to design a kernel that can be clocked faster even if this causes loops to be pipelined with longer initialization intervals. One future work could be to investigate if kernels and memory can be clocked separately. In such a situation, new optimization opportunities may arise that can potentially try to clock the memory as high as possible while minimizing the initialization interval of kernels.

The performance difference between BFS and DOBFS proved that algorithmic optimizations are still relevant for specialized hardware and abstract workflows such as HLS. BFS implementation did not show a substantial performance improvement since it continues to be limited by the memory bandwidth. On the other hand, DOBFS provided significant performance improvements with the help of the decreased memory load.

The performance difference between BFS and CC proved that overheads continue to be a significant problem. Communications between the CPU and the FPGA, memory copy operations, and commands such as queuing kernels are costly in terms of performance. Thus, it is worth investigating if the tasks of the host program can be offloaded to FPGA as well.

Chapter 8

Conclusion

Our aim in this thesis is to propose techniques to execute graph applications on heterogenous CPU+FPGA architectures such as Intel's Xeon-Arria platform. We opted for developing task kernels in OpenCL which are synthesized to hardware by the HLS workflow.

We surveyed algorithmic optimizations that can be employed for iterative graph algorithms PR, BFS, and CC, specifically. We also investigated HLS optimizations that enable the synthesis of well pipelined, performant kernels.

We combined algorithmic and HLS optimizations to implement a range of OpenCL kernels that have different features and optimizations. Then, we tested them with a range of graphs to demonstrate the effect of implemented optimizations. To exploit the parallelization capabilities, we developed a methodology that traverses graphs that are stored in the edge list and CSR format in a well-pipelined, efficient way.

We presented our findings which provides insights about how to optimize the hardware without directly interacting with it at the HDL/RTL level. We discussed which optimizations are favored by the HLS process within the context of iterative graph algorithms.

In conclusion, it is possible to execute graph applications faster by utilizing the HLS process and the underlying FPGA. Specifically, we see an average of 2.5X improvement in our implementations when compared to the baseline.

Bibliography

[1] D. F. Carr, "How google works," *Baseline Magazine*, vol. 6, no. 6, 2006.

[2] J. Clement, "Hours of video uploaded to youtube every minute as of may 2019. statista," 2019.

[3] I. Mansaku, S. Mansaku, and I. Tampakoudis, "An empirical comparison of the major stock exchanges: Nyse, nasdaq and lse in perspective," *Academic Journal of Interdisciplinary Studies*, vol. 5, no. 3 S1, p. 406, 2017.

[4] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, *et al.*, "Intel nehalem processor core made fpga synthesizable," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 3–12, 2010.

[5] C. Andriamisaina, E. Casseau, and P. Coussy, "Synthesis of multimode digital signal processing systems," pp. 318 – 325, 09 2007.

[6] S. Beamer, K. Asanović, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 820–831, IEEE, 2017.

[7] K. Lakhotia, R. Kannan, and V. Prasanna, "Accelerating pagerank using partition-centric processing," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 427–440, 2018.

[8] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC'12: Proceedings of the International Conference on High*

Performance Computing, Networking, Storage and Analysis, pp. 1–10, IEEE, 2012.

[9] G. M. Slota, S. Rajamanickam, and K. Madduri, "Bfs and coloring-based parallel algorithms for strongly connected components and related problems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 550–559, IEEE, 2014.

[10] M. Sutton, T. Ben-Nun, and A. Barak, "Optimizing parallel graph connectivity computation via subgraph sampling," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 12–21, IEEE, 2018.

[11] J. Soman, K. Kishore, and P. Narayanan, "A fast gpu algorithm for graph connectivity," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–8, IEEE, 2010.

[12] J. Definelicht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[13] I. Altera, "Intel fpga opencl sdk best practice guide," *URL https://www. altera. com/enUS/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide. pdf*.

[14] A. S. Altera, "for opencl programming guide, 2013," *Dostopno na: https://www. altera. com/content/dam/altera-www/global/en US/pdfs/literature/hb/opencl-sdk/aocl programming guide. pdf.[Dostopano 5.3. 2016]*.

[15] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 409–420, IEEE, 2016.

[16] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, IEEE, 2014.

[17] R. Domingo, R. Salvador, H. Fabelo, D. Madroñal, S. Ortega, R. Lazcano, E. Juárez, G. Callicó, and C. Sanz, "High-level design using intel fpga opencl: A hyperspectral imaging spatial-spectral classifier," in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, IEEE, 2017.

[18] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark: A high-level synthesis framework for applying parallelizing compiler transformations," in *16th International Conference on VLSI Design, 2003. Proceedings.*, pp. 461–466, IEEE, 2003.

[19] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," tech. rep., Stanford InfoLab, 1999.

[20] A. Bundy and L. Wallen, "Breadth-first search," in *Catalogue of artificial intelligence tools*, pp. 13–13, Springer, 1984.

[21] L. Di Stefano and A. Bulgarelli, "A simple and efficient connected components labeling algorithm," in *Proceedings 10th International Conference on Image Analysis and Processing*, pp. 322–327, IEEE, 1999.

[22] P. Caldeira, J. C. Penha, L. Bragança, R. Ferreira, J. A. M. Nacif, R. Ferreira, and F. M. Q. Pereira, "From java to fpga: An experience with the intel harp system," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 17–24, 2018.

[23] A. M. Cabrera and R. D. Chamberlain, "Exploring portability and performance of opencl fpga kernels on intel harpv2," in *Proceedings of the International Workshop on OpenCL*, pp. 1–10, 2019.

[24] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science Business Media, 2012.

[25] R. Yates, "Fixed-point arithmetic: An introduction," *Digital Signal Labs*, vol. 81, no. 83, p. 198, 2009.

[26] B. Brewer, "libfixmath." `https://github.com/PetteriAimonen/libfixmath`, 2012.

[27] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[28] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *IN SPAA*, pp. 233–244, 2009.

[29] A. M. Cabrera, C. J. Faber, K. Cepeda, R. Derber, C. Epstein, J. Zheng, R. K. Cytron, and R. D. Chamberlain, "Dibs: A data integration benchmark suite," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 25–28, 2018.

[30] D. Manual, "Intel 64 and ia-32 architectures software developers manual," 2016.

[31] P. K. Gupta, "Accelerating datacenter workloads," in *26th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 2017, p. 20, 2016.

[32] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *Proceedings of the 4th ACM European conference on Computer systems*, pp. 205–218, 2009.

[33] S. Kulkarni, A. Singh, G. Ramakrishnan, and S. Chakrabarti, "Collective annotation of wikipedia entities in web text," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 457–466, 2009.

[34] P. Erdos and A. Reyni, "On random graphs," *Publicationes Mathematicae*, 1959.

[35] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in *European conference on principles of data mining and knowledge discovery*, pp. 133–145, Springer, 2005.

[36] "Graph500 benchmark. www.graph500.org."

[37] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, pp. 442–446, SIAM, 2004.

[38] D. A. Bader and K. Madduri, "Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks," in *2008 IEEE international symposium on parallel and distributed processing*, pp. 1–12, IEEE, 2008.

[39] E. N. Gilbert, "Random graphs," *The Annals of Mathematical Statistics*, vol. 30, no. 4, pp. 1141–1144, 1959.

[40] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?," in *Proceedings of the 19th international conference on World wide web*, pp. 591–600, 2010.

Appendix A

Code

A.1 An example implementation of single PageRank kernel

```
inline float mul(float f1, float f2)
{
   return f1*f2;
}
struct tuple{
   bool is_zero_in;
   unsigned next_offset;
};
struct tuple2{
   unsigned dvid;
   float score;
};
//channel unsigned chan1;
#pragma OPENCL EXTENSION cl_altera_channels : enable
channel struct tuple chan0;
channel struct tuple chan1;
channel struct tuple chan2;
```

```
channel struct tuple chan3;

channel struct tuple2 pr0;
channel struct tuple2 pr1;
channel struct tuple2 pr2;
channel struct tuple2 pr3;

__attribute__ ((task))
__kernel void producer0(

__global const unsigned* restrict offsets, //0
unsigned const start_of_dvid, //1
unsigned const end_of_dvid //2
)
{
   for(unsigned dvid = start_of_dvid; dvid < end_of_dvid ; dvid++){
      struct tuple t;

      t.next_offset = offsets[dvid+1];


      if(offsets[dvid] == offsets[dvid+1])
         t.is_zero_in = true;
      else
         t.is_zero_in = false;


      write_channel_altera(chan0, t);

   }
}
__attribute__ ((task))
__kernel void consumer0(

__global float* restrict pg_val_next, //0
```

```
53 unsigned const start_of_dvid, //1
54 unsigned const end_of_dvid //2
55 )
56 {
57    for(unsigned dvid = start_of_dvid; dvid < end_of_dvid ; dvid++)
58    {
59       struct tuple2 t = read_channel_altera(pr0);
60       pg_val_next[t.dvid] = t.score;
61       //printf("consumer: pg_val_next[%u] = %f\n",t.dvid,t.score);
62    }
63 }
64
65 __attribute__ ((task))
66 kernel void compute_PR0(
67
68 __global const float* restrict pg_val, //0
69 __global const float* restrict pg_division, //1
70 __global const unsigned* restrict offsets, //2
71 __global const unsigned* restrict edges, //3
72
73 float const dampener, //4
74 float const dampen_av, //5
75 unsigned const start_of_dvid, //6
76 unsigned const end_of_dvid //7
77
78 //deltas //8
79 //__global uint* restrict pg_val_next //9
80 )
81 {
82       float temp;
83    float toAdd = 0;
84    unsigned dvid = start_of_dvid; //destination vertex id
85    unsigned j = offsets[start_of_dvid];
86
87
```

```
88    bool next_dvid = true;
89    bool is_zero_in;
90    unsigned next_offset;
91
92    while( dvid < end_of_dvid) // edge loop
93    {
94       if(next_dvid)
95       {
96          struct tuple t = read_channel_altera(chan0);
97          is_zero_in = t.is_zero_in;
98          next_offset = t.next_offset;
99          next_dvid = false;
100      }
101      if ( is_zero_in )
102      {
103         //pg_val_next[dvid] = mul(dampen_av, pg_division[dvid]);
104         temp = mul(dampen_av, pg_division[dvid]);
105         //dvid++;
106         next_dvid = true;
107      }
108      else if( next_offset == j+1 )
109      {
110         toAdd += pg_val[edges[j]];
111         //pg_val_next[dvid] = mul( (dampen_av + mul(dampener, toAdd))
               , pg_division[dvid]   );
112         temp = mul(  (dampen_av + mul(dampener, toAdd)) ,
               pg_division[dvid]    );
113         toAdd = 0;
114         //dvid++;
115         next_dvid = true;
116         j++;
117      }
118      else
119      {
120         toAdd += pg_val[edges[j]];
```

```
121           j++;
122       }
123
124       if(next_dvid)
125       {
126           struct tuple2 t;
127           t.dvid = dvid;
128           t.score = temp;
129           write_channel_altera(pr0, t);
130           //printf("PR: pg_val_next[%u] = %f\n",dvid,temp);
131           dvid++;
132       }
133   }
134
135 } //kernel
```

Listing A.1: An example implementation of single PageRank kernel (floating-point, producer, flat loop).