SOURCE-TO-SOURCE TRANSFORMATION BASED METHODOLOGY FOR GRAPH-PARALLEL FPGA ACCELERATORS

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
Cemil Kaan Akyol
August 2019

Source-to-Source Transformation Based Methodology for Graph-Parallel FPGA Accelerators
By Cemil Kaan Akyol
June 2019

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)

M. Mustafa Özdal

Süleyman Tosun

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

SOURCE-TO-SOURCE TRANSFORMATION BASED METHODOLOGY FOR GRAPH-PARALLEL FPGA ACCELERATORS

Cemil Kaan Akyol
M.S. in Computer Engineering
Advisor: Özcan Öztürk
August 2019

Graph applications are becoming more and more important with their widespread usage and the amounts of data they deal with. Biological and social web graphs are well-known examples which show the importance of efficient processing of the graph analytic applications and problems. Addressing those problems in an efficient manner is not a straightforward task. Distributing and parallelizing the computation, and integrating hardware accelerators are the main approaches that were tried during the last decade. However, these approaches mainly focus on specific legacy algorithms and may not completely solve the problems. Therefore, when there is an emerging need for a non-legacy algorithm targeting a specific problem, the developer has to cope with the adversaries of the distribution, parallelization techniques, and hardware specifications to parallelize and accelerate the application. Our proposed source-to-source based methodology gives the freedom of not knowing the low-level details of parallelization and distribution by translating any vertex-centric C++ graph application into pipelined SystemC model. In order to support different types of graph applications, we have implemented several features like non-standard application support, active set functionality, multi-pipeline support, etc. The generated SystemC model can be synthesized by High-Level Synthesis (HLS) tools to obtain the FPGA programming image, i.e., the bitstream. Our accelerator development flow can generate two different execution models, high-throughput (HT) and work-efficient (WE). Compared to OpenCL counterparts of the algorithms, HT and WE models perform slightly better in terms of execution time and throughput. WE model performed approximately 40% better than OpenCL in terms of work done and execution time. Therefore, the proposed source-to-source based methodology is able to provide more efficient hardware designs by only requiring a simple high-level language description from the user.

Keywords: Source-to-Source Transformation, Hardware Accelerators, FPGA, Active Set, Asynchronous Execution.

ÖZET

KAYNAKTAN KAYNAĞA DÖNÜŞÜME DAYALI PARALEL ÇİZGE FPGA HIZLANDIRICILARI YÖNTEMİ

Cemil Kaan Akyol
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Özcan Öztürk
August 2019

Çizge uygulamaları, yaygın kullanım alanları ve ele aldıkları veri miktarları ile gittikçe daha fazla önem kazanmaktadır. Biyolojik ve sosyal web çizgeleri, çizge analitik uygulamalarının ve problemlerinin verimli işlenmesinin önemini gösteren bilinen örneklerdir. Bu sorunların verimli bir şekilde ele alınması kolay bir iş değildir. Hesaplamanın dağıtılması ve paralel hale getirilmesi ve donanım hızlandırıcılarının eklenmesi, son on yılda denenen ana yaklaşımlardır. Bununla birlikte, bu yaklaşımlar temel olarak belirli eski algoritmalara odaklanır ve sorunları tamamen çözemeyebilir. Bu nedenle, belirli bir sorunu hedefleyen yeni bir algoritmaya ihtiyaç duyulduğunda, geliştirici, uygulamayı paralelleştirmek ve hızlandırmak için dağıtım, paralelleştirme tekniklerinin ve donanım özelliklerinin üstesinden gelmek zorundadır. Önerilen kaynaktan kaynağa temelli metodolojimiz, düğüm merkezli herhangi bir C++ çizge uygulamasını boruhattı bazlı SystemC modeline çevirerek paralellik ve dağıtımın düşük seviyeli ayrıntılarını bilmeme özgürlüğünü verir. Farklı çizge uygulama türlerini desteklemek için standart dışı uygulama desteği, etkin set fonksiyonu, çoklu boru hattı desteği gibi çeşitli özellikleri uyguladık. Üretilen SystemC modeli, Üst Düzey Sentez (HLS) araçları ile sentezlenebilir; FPGA programlama görüntüsü, yani bitstream oluşturabilir. Hızlandırıcı geliştirme akışımız, yüksek verimli (HT) ve iş verimli (WE) olmak üzere iki farklı uygulama modeli üretebilir. Algoritmalar OpenCL benzerleri ile karşılaştırıldığında, HT ve WE modellerinin yürütme süresi ve verimi bakımından biraz daha iyi performans gösterdiği görülmektedir. WE modeli, yapılan iş ve uygulama süresi açısından OpenCL'den yaklaşık % 40 daha iyi performans göstermiştir. Bu nedenle, önerilen kaynaktan kaynağa temelli metodoloji, kullanıcıdan sadece basit bir üst düzey dil tanımı gerektirerek daha verimli donanım tasarımları sağlayabilmektedir.

Anahtar sözcükler: Kaynaktan Kaynağa Dönüşüm, Hızlandırıcı Donanımlar, FPGA, Etkin Set, Eş Zamanlı Olmayan İşleme.

Acknowledgement

This thesis is the ultimate step of 3 intense years and innumerable sleepless nights in obtaining my M.S. degree. By finishing this journey, I would like to thank all the people who does not refrain their support and faith in me during my studies.

First and foremost, I would like to express my deepest appreciation to my supervisor Prof. Dr. Özcan Öztürk for his constant support, kindness, and encouragement during my entire graduate study process and providing me the opportunity to work with him. I consider myself much lucky to have a supervisor like him.

I would like to express my sincere thankfulness to Assoc. Prof. Dr. M. Mustafa Özdal for guiding me to solve the problems and helping me find a way out of many dead ends. Without his support and guidance, I would not complete this thesis.

The completion of my thesis would not have been possible without the support and nurturing of my parents, Mehmet and Hatice, and my lovely sister İrem. I am very grateful to my father who always stands beside me. Being a father like him seems impossible to me, but I will do my best as I am learning from him. I am also very grateful to my mother for her unending love and patience. I am blessed to have been born to a great mother. Together, they always tried to present me and my sister better life and to raise us as good individuals by overcoming many difficulties and working hard. I am forever indebted to them for being an incredible family.

I must express my gratitude to my beloved, Sevde. She was always by my side, cheered me up and comforted me, helped me grow emotionally, supported me when I feel anxious, distressed. I am lucky to have a companion like her.

I am also grateful to Ebru Ateş, for her endless kindness and help during my 8-year Bilkent life. Her door is always open to me whenever I need help.

Finally, I would like to thank my friends, Göktuğ Mert, Sevil Yaşar, Murateren

İlgar, Caner Mercan, Sinem Sav for their friendship, collaboration, and support. They made me have a lot of fun.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The amount of data produced and stored per year grows in multi folds with the advancements in Internet technologies, smart devices, and cloud services. One form of storing and managing the data is to use graphs such as social networks and web graphs. Analyzing these large graphs is done for different domains like Bioinformatics, Machine Learning, Data Mining (MLDM), where there can be millions of nodes with billions of connections between.

Many algorithms with different objectives are designed to analyze and process those graphs. For example, PageRank [1] is a state-of-the-art ranking algorithm calculating the vertices' importance. Similarly, Breadth First Search (BFS) is a graph traversal algorithm, to search vertices. While there are many other important graph applications that target different problems, their common property is that they suffer in terms of memory footprint and serial execution time. However, the amount of data they need to operate on continuously increases.

Therefore, there is a great need for efficient processing of this rapidly growing graph data. Improving the efficiency and optimizing the application may provide huge performance gains, which in turn, will help the researchers and the developers. Thus, graph data processing started to gain more and more attention.

The most straightforward approach for graph processing is the parallelization

of applications by splitting the execution into multiple processors or machines. Parallel computing can drastically reduce the runtime in theory, however, the main bottleneck is the communication. More specifically, communication between the processors and the communication required for memory accesses is critical. If the application is not implemented efficiently, the runtime may not improve at all.

Moreover, parallel programming is not an easy task considering the synchronization requirements, potential race conditions, irregular memory accesses, and load balancing problems. Therefore, it requires background knowledge, experience in implementation with deeper understanding. Especially, doing this at the hardware level means a great deal of learning process and a huge investment in resources and time.

There have been many efforts to make it easier to implement a parallel application while increasing the efficiency. Software efficiency improvement, source-to-source transformation, automatic parallelization, and hardware acceleration are some of many techniques used in the literature. This thesis tries to combine these concepts to create a source-to-source transformation based methodology for FPGA acceleration.

We believe improving the user experience and enhancing the efficiency will have a great impact in this domain due to wide spread usage and growing need.

1.1 Contributions

Given a simple high level language (C++) description of an application, our accelerator design flow generates the final hardware accelerator ready to be embedded into the FPGA. During this process, the proposed application flow 1) creates an intermediate representation of a vertex-centric graph application implemented in C++ by extracting data from Abstract Syntax Tree (AST), 2) generates a SystemC model using intermediate representation with the help of template-based

methodology [2], 3) creates RTL design of the generated SystemC model using High Level Synthesis (HLS) flows, and 4) generates bitstream ready to be used on FPGA boards.

Compared to other methodologies on graph parallel application acceleration which will be discussed in detail in Chapter 2, our model widens the application types and increases the efficiency by integrating source-to-source transformation. With the help of this integration, the inexperienced users in hardware accelerators and parallel applications can easily accelerate their applications, without dealing with the low-level details like synchronization, race conditions, or hardware usage.

This thesis shows the lifetime of a graph application in our flow, from plain C++ code all the way to the FPGA board. Therefore, different support mechanisms and improvements in application development flow constitutes our main contributions which can be briefly described as follows:

- Ease of use: The development process of hardware accelerators demands huge investment both in time and resources. Moreover, the core need is deep knowledge of parallel execution and being able to deal with hardware specifications and descriptions. These requirements are not usually available for average developer who wants to speed up their long-lasting graph-applications. On the other hand, without one of the above requirements, it is not straightforward to design and develop an accelerator. Even if the accelerator is developed, it will most likely be an inefficient one.

  Therefore, a key contribution provided by the source-to-source transformation based accelerator design flow is enabling the developer without dealing with the low-level details, thereby saving time and manpower.

- Asynchronous execution support: For graph accelerators, asynchronous execution is one of the major advancements in terms of efficiency. In synchronous execution, there are strict barriers between the iterations, to avoid the data dependency problems and provide synchronization. However, eliminating barriers improves the performance. Hence, with the help of local

data structures to circumvent data dependency problems and handle synchronization, we support asynchronous execution in our model.

- Active set: In many of the graph applications, a set of vertices do not require to be processed during the lifetime of the application, that is, they may converge earlier than the others, or the termination condition may be met, etc. This vertex set can be discarded from the next iteration, thereby reducing the amount of work to do. This, in turn, will result with a reduction in both time complexity and power consumption. However, for some applications, it is necessary to process all the vertices until they all converge. For such applications, our implementation will be as good as other designs.

  In order to meet these different execution requirements, we implemented two different models, namely, work-efficient and high-throughput. The work-efficient model operates only on the active vertices, whereas the high-throughput version executes for all the vertices.

- Extended features: There are wide variety of graph applications, which may require the developer to implement different features beyond the generic gather, apply, scatter (GAS) functions. To support different graph algorithms, our accelerator design flow includes various features such as conditional iteration over the neighbor edges, supporting non-GAS applications, enabling non-neighbor data access, multiple vertex program support, and providing user-defined types. With these additional features almost all of the vertex-centric graph parallel applications can be modelled by our accelerator design flow.

1.2 Outline

The remainder of this thesis is structured as follows. Next chapter gives a detailed discussion of the related work on automatic parallelization, improving efficiency

of software applications, source-to-source transformation, and hardware accelerators. Background information on vertex centric graph processing, Gather-Apply-Scatter (GAS) programming abstraction model, synchronous vs asynchronous execution, source-to-source transformation, and The Hardware Accelerator Research Program (HARP) is given in Chapter 3. Chapter 4 presents our approach in two main parts: the high-level view of our application flow and the programming model. Source to source transformation details are given in Chapter 5. Our accelerator details with supported features are discussed in Chapter 6. Chapter 7 presents the experimental setup, architectural settings, platforms used, graph applications tested for evaluations, and the results from our experiments. Chapter 8 concludes the thesis with a summary of our major observations.

Chapter 2

Related Work

There are many research directions to improve performance and efficiency of computing systems including automatic parallelization of applications, source-to-source transformations, distributed computing, and hardware accelerators. We introduce these approaches and will discuss different techniques in these domains that are relevant for our approach on the field.

From automatic parallelization perspective, there are various studies exploring different directions. For example, Polaris compiler [3] is an automatic tool to parallelize and optimize the loops in sequential Fortran programs. Similarly, SUIF [4] is a multi-language compiler that involves a set of development tools and supports automatic loop level parallelism and optimization. Liao et al. [5] automatically parallelize C++ applications using a multiple-language source-to-source compiler that preserves high-level abstractions. There are also commercial compilers like Intel C++/Fortran compiler [6], which is focused on vectorizing the loops by using SIMD (Single-Instruction-Multiple-Data) parallelism with OpenMP pragmas. Our work is not targeting loop level parallelism, but rather trying to parallelize the whole application by creating a pipeline structure while maintaining synchronization between the modules.

Our contribution is not limited to the source-to-source transformation, but it is

the first and foremost stage of our work. In previous studies, source-to-source transformation is used for different goals. For example, Togpu [7] and GPSME [8] are source-to-source transformation tools that converts C++ program into CUDA program. The Vienna Fortran Compiler [9] is also a source-to-source transformation and parallelization system that translates Fortran95 programs to Fortran90 programs. This tool is similar to our work in terms of translation. It consists of several modules, the first one being data collection using Abstract Syntax Tree (AST) and the next one to generate the target-language code using collected data. However, this work does not target hardware acceleration, it is purely a software-level parallelization and optimization tool.

There are several studies on manipulating and using program AST and analyzing the application beyond the source-to-source transformation, like easing parallelization phase, generating some helper visuals, etc. Duffy et al. [10] develop a tool using Clang Parser [11] to compute some metrics of code complexity. Schmidt et al. [12] generate thread communication graphs from SystemC source code to help the developers and system designers in understanding the libraries, legacy codes. Pina VM [13] is a SystemC front-end that retrieves structural information of the application. Chen et al. [14] implemented a tool to detect possible race conditions and synchronization failures which arise from shared variable usage during parallel execution. Systemc-clang [15] is a static analyzer that can identify communication structure in a SystemC model. Togpu [7] uses AST to identify code sections of interest during automatic parallelization. Compass [16] uses AST to detect software bugs with a recursive tree visitor function. In our work, we created a tool similar to these which extract the meta-data and sections of interest about the user's C++ code and analyze it. Moreover, shared variable usage, communication scheme across threads, read-write ports to memory subsystem are extracted from input code using AST.

There are also lots of efforts for optimizing and accelerating graph applications. Widely used technique is to distribute a large-scale graph across multiple machines and processors. Well-known multi-purpose distributed software frameworks can be found in the literature like Pregel [17], GraphLab [18], and MapReduce [19]. There are many extensions to these frameworks, however, their main

goal is to provide an easy to use interface and to improve iterative application performance.

In addition to software techniques, there are also hardware based approaches to process big data problems. There are several hardware resources used for this purpose including Graphics Processing Units (GPUs), Application Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs). These are used to accelerate a wide set of applications, such as deep learning-neural networks [20][21][22], bioinformatics [23][24][25], graph applications [2][26][27], and cryptography [28].

To accelerate a given application on GPU, there are two different approaches. The first approach is to use source-to-source transformation to generate CUDA code. Like in Togpu [7], using only plain C++ code with some limitations, one can generate CUDA code and accelerate the design on GPUs. The other approach is directly writing CUDA application such as Manavski et al. [28] and Nurvitadhi et al. [21]. The drawback of this approach is that the developer has to be experienced in implementing parallel applications, since there are low-level details like race conditions and data dependencies. Without deeper knowledge of these concepts, the application will likely be inefficient.

ASIC is an integrated circuit produced for a particular use, with a single functionality. Once it is designed and manufactured, it cannot be changed. Therefore, it is not considered as a general-purpose hardware. For example, Samba [23] uses ASIC as hardware accelerator alongside an FPGA. Similarly, Nurvitadhi et al. [21] creates an ASIC accelerator for benchmarking purposes.

FPGAs are general-purpose chips that execute some set of algorithms to accelerate the applications. On the contrary to ASIC chips, no special design and production process is needed and can be reconfigured easily. Therefore, FPGAs attract the community's attention more than ASICs. There are lots of research efforts on FPGA accelerators, since the other accelerators do not fulfill the requirements of the community. For our specific graph-parallel application acceleration domain, McGettrick et al. [26] develop an FPGA accelerator for

Pagerank eigenvector problem. Cygraph [27], Betkaoui et al. [29], Umuroglu et al. [30], Wang et al. [31] have implemented several variations of breadth-first-search algorithms on FPGA. Jagadeesh et al. [32] have also created FPGA accelerator for the single-source-shortest-path algorithm. There are also more generic application frameworks, which can support multiple applications [33][34]. Even some of these frameworks can execute multiple algorithms, they cannot implement non-legacy applications. For example, these works cannot handle irregular graph applications in which there are lots of irregular memory accesses and asynchronous execution.

Application development process on GPU is much easier than FPGA and ASIC. Moreover, GPUs are more accessible, thereby having more impact and wider usage in the community. On the other hand, FPGAs and ASICs with their limited resources and bandwidths, can perform similar to GPUs while using lower energy.

It is shown that FPGA and ASIC implementations perform similarly [21]. In this evaluation, they used different metrics for comparison, which shows that ASIC implementations have slightly better results. However, ASIC chips are customized for a particular use, so without long term execution and investment, it is not viable to use them for accelerating generic applications.

Similar to our work, Ayupov et al. [2] addresses irregular graph application properties by creating a template-based design which supports active-vertex-set and asynchronous execution. More specifically, user can customize the application by providing user functions as an input to the design. However, there are still some missing features that cannot be handled in this kind of template based design. First, the user does not have the flexibility to create a wide range of applications, because, the design specifically focuses on Gather-Apply-Scatter (GAS) applications. Moreover, user needs to change the data structures in the SystemC application to be able to implement an efficient solution. In this work, we extend the template based design by providing source-to-source transformation, thereby, allowing the user to write plain C++ code. With this improvement, template-based design for both GAS and non-GAS applications will automatically be generated.

Chapter 3

Background

3.1 Vertex-Centric Graph Processing

Traditional implementations of the graph algorithms consist of iteration over the vertices and edges, using data structures to hold the data about those containers, such as Dijkstra's algorithm [35] in which priority queue is used. In such algorithms, there is a broader goal while accessing all the data structures, edges, and vertices.

However, in vertex-centric graph processing, the applications are implemented using a single vertex point of view, that is why those type of applications have a *Vertex Program*. Since the execution of that program takes place for each vertex, it can be expressed as "Think-Like-A-Vertex" [17]. *Vertex Program* can read the neighbor data, send or receive data using channels, update and change the local data. In our accelerator flow, the application that is provided by the user should be implemented in vertex-centric execution model.

3.2 Gather-Apply-Scatter

Gather-Apply-Scatter (GAS) is a programming abstraction model presented by PowerGraph [36], where graph problems are modelled in a "Think-Like-A-Vertex" [17] manner. In such a scenario, computation for a graph is distributed on a cluster to address the bottlenecks caused by high degree vertices. This is primarily achieved by parallelizing the application over the edges. The GAS model consists of three separate phases, where *Gather* phase collects the information over the neighbor vertices. Using the edges, neighbor data is collected and accumulated to construct the vertex data to be used by the processed vertex. Then, in the *Apply* phase, the value in the vertex is updated using the accumulated data in the Gather phase. *Scatter* is the last phase which informs the neighbor vertices and activates them by using the value of the vertex that is currently being processed. There are also variations of the GAS model, such as having an additional phase *Add* which is about gathering the neighbor data and collecting into a data segment using a function. Furthermore, for some applications *Scatter* phase is not mandatory [37]. Note that, applications modelled with GAS abstraction needs the user to specify those methods explicitly [2][37].

3.3 Synchronous vs Asynchronous Execution

Synchronous execution uses well-defined iterations with barriers in between. In graph applications, synchronous execution allows neighbor data calculated at iteration *t-1* to be used at iteration *t*. On the contrary, in asynchronous execution, well-defined iterations are eliminated, allowing the most recent data to be accessed anytime. As explained in the literature, there are many problems which could be solved either synchronously or asynchronously. Linear systems [38], belief propagation [39], expectation maximization[40], Pagerank [41] [18], and stochastic optimization [42][43] are some examples where synchronous execution converges much slower than the asynchronous execution. More importantly, asynchronous execution reduces the total work done. For example, for PageRank algorithm [44]

this reduction is about %30.

3.4 Source-to-Source Transformation

A compiler that is capable of generating equivalent target programming-language implementation using source code is called source-to-source compiler. Target programming language may be the same as source code language or a completely different language. Unlike traditional compilers, the source-to-source compiler does not translate higher-level application to the lower-level, e.g. Java to bytecode. It translates the source languages into target languages that use the same level of abstraction. For example, converting C++ applications into CUDA program is source-to-source transformation. In our accelerator flow, C++ application is translated into equivalent SystemC application.

3.5 Hardware Accelerator Research Program

The Hardware Accelerator Research Program (HARP) funded by Intel provides faculty members and researchers different programming tools, operating systems with Xeon Processors and FPGA systems, globally. The ultimate aim of this program is speeding up the researches on accelerator-based computing systems. This program also provides tutorials, technical support, etc. [45]. Our template-based FPGA accelerators are generated with the provided synthesis tools and executed on these computer clusters with the FPGA systems.

Chapter 4

Our Approach

4.1 High Level View

The proposed design flow is shown in Figure 4.1. As can be seen, different parts of the design flow is separated by color codes. Specifically, light grey color represents the user code, white color represents the tools, libraries, and implementations, whereas, the dark grey color corresponds to automatically generated models in the flow.

Our approach starts with the user code, on which we perform a series of operations to translate it into the bitstream. User can write any type of graph application with a wide set of features. As will be explained later, user can iterate over the neighboring vertices a different number of times (Sec. 6.2), activate all neighbors or some (Sec. 6.3), put conditionals (Sec. 6.4), access the data of any vertex (Sec. 6.5) or write multiple applications and use them (Sec. 6.6).

Front-end parser and code generation tool are the tools that we implemented where source-to-source transformation takes place. The parser is implemented to search for *VertexInfo* requests, incoming or outgoing *EdgeInfo* requests, neighbor *VertexInfo* requests, neighbor and self *VertexData* requests, *LocalVertexData*

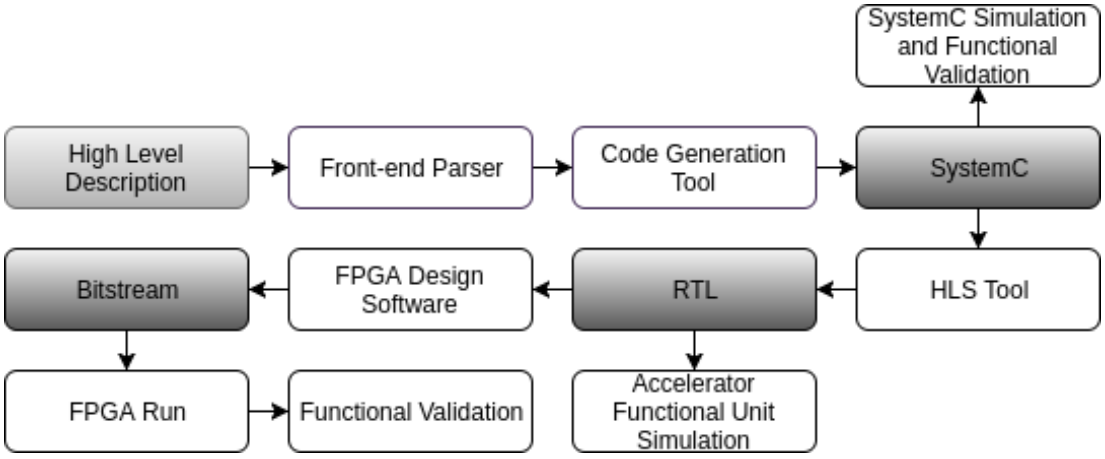

Figure 4.1: High level view of our approach from C++ to Bitstream. Color-coding: light grey — user-code, dark grey — auto-generated models, white — used tools and libraries.

updates, conditional flows, variables used across the application, data types and user-defined types, *VertexData* fields which can be shared among neighbor vertices or private, etc. After collecting meta-data about the application, the application is divided into smaller segments called threads. Using the data flow across those threads, FIFOs are created. With the data flow and *VertexData* fields, global tables are generated. Therefore, using those FIFOs, threads, and meta-data, SystemC files are generated with small additions like taking input from and sending to output FIFOs, reading from global tables, etc.

SystemC is the result of the source-to-source transformation. It is a widely used hardware description language which uses a set of C++ classes and provides modeling and simulation interface. The generated hardware description consists of modules, namely, threads, inter-module FIFOs. In Figure 4.2 (see Section 4.2.4 for further details), the basic architecture of a simple graph application is given. These modules do not only contain the code snippets from the user code, but also contain some helper template structures [2] that handle the communication between the accelerator unit and the memory interface which contains read and write requests and responses, etc.

Generating SystemC code from C++ user code using the front-end parser and

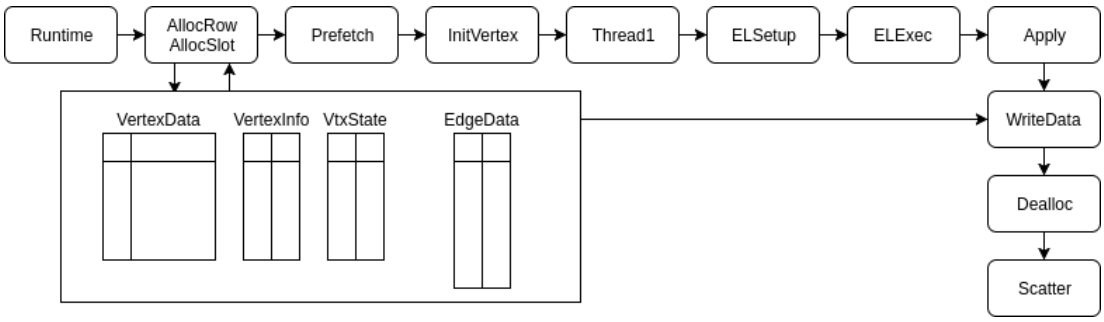

Figure 4.2: Basic architecture of a graph application implemented in SystemC.

code generation tool is the main contributions of this thesis. The rest of the accelerator flow involves generating intermediary-steps, validation, and verification.

SystemC simulation and functional validation steps are about comparing and checking the results, whether generated parallel SystemC model and the serial version implemented in C++ are matching.

Once the SystemC model is simulated and validated, RTL is generated using an HLS tool. During this flow, timing characterizations are extracted like latency and throughput. Using these values, the system-level performance model is automatically produced for the accelerator. Produced model is then used for design space exploration [2]. The synthesis of SystemC models to generate RTL is handled by a standard HLS flow after the automatic template parameter tuning using system-level performance model and the design space exploration.

High-Level-Synthesis tools [46][47] generally accept synthesizable subsets of C/C++ and MATLAB, and perform source-to-source compilation to generate an RTL design. It is a design abstraction to model the digital circuits in which signals (data) flow across hardware registers and arithmetic operations are performed on those signals (data). Once the RTL design is generated, it can be used as a higher-level abstraction of the circuit or lower-level representations, in which one can see actual components of the circuit and wiring can be derived.

The aim of accelerator functional unit simulation is to verify that RTL is generated without errors and designed correctly. RTL communicates with the hardware

interface of the simulation environment which pretends to be the FPGA. On the other hand, the host code communicates with the software interface. When simulation finishes, verification of the system is handled by comparing the results of RTL design and the host software. If there are no mismatches, it can be said that RTL generation is successful and most likely the bitstream generation will be successful as well since the simulation environment mimics FPGA using the RTL.

FPGA Design Software is a logic synthesis tool [48] and synthesizes RTL designs. The design that is generated by HLS tool and tested by accelerator functional unit simulation is loaded to FPGA design software. After running a series of algorithms on that, a device programming image is produced which can be loaded and run on FPGA.

Once synthesis is completed and meets timing and resource constraints, the compiled programming image can be executed on FPGA. With bitstream, the host code is also executed and once again, the results of bitstream execution and host code execution are compared. If there are no mismatches, then it can be concluded that, the bitstream is generated correctly. The ultimate aim of the whole application flow is to generate the bitstream and verify that it runs correctly when compared with the serial version of the application which is the user's source code in C++.

4.2 Programming Model

4.2.1 Clang

We have implemented a Clang [11] plugin to collect and extract the meta-data about the C++ application provided by the user. Clang is a C/C++ compiler that provides many useful source level tools and open-source LLVM front-end [49]. It is widely preferred in industry for fast compilation, better error reporting, and expressive diagnostics. Moreover, there are many additional tools such as Clang

```
1
2 ForStmt 0x37982d0
3 |-DeclStmt 0x3797f00
4 | `-VarDecl 0x3797d68  used eItr 'class EdgeIterator' cinit
5 |   `-ExprWithCleanups 0x3797ee8 'class EdgeIterator'
6 |     `-CXXConstructExpr 0x3797eb0 'class EdgeIterator' 'void (const class EdgeIterator &)
      throw()' elidable
7 |       `-MaterializeTemporaryExpr 0x3797e98 'const class EdgeIterator' lvalue
8 |         `-ImplicitCastExpr 0x3797e80 'const class EdgeIterator' <NoOp>
9 |           `-CXXMemberCallExpr 0x3797e50 'class EdgeIterator'
10 |             |-MemberExpr 0x3797df0 '<bound member function type>' .beginEdgeIterator 0
      x37965a0
11 |             | `-DeclRefExpr 0x3797dc8 'class VertexHandle' lvalue ParmVar 0x37979c8 'vtx' '
      class VertexHandle &'
12 |             `-DeclRefExpr 0x3797e28 'EdgeType' EnumConstant 0x3795e60 'IN_EDGE' 'EdgeType'
13 |-<<<NULL>>>
14 |-UnaryOperator 0x3797fb8 '_Bool' prefix '!'
15 | `-CXXMemberCallExpr 0x3797f78 '_Bool'
16 |   `-MemberExpr 0x3797f40 '<bound member function type>' .isEnd 0x3795030
17 |     `-ImplicitCastExpr 0x3797fa0 'const class EdgeIterator' lvalue <NoOp>
18 |       `-DeclRefExpr 0x3797f18 'class EdgeIterator' lvalue Var 0x3797d68 'eItr' 'class
      EdgeIterator'
19 |-CXXOperatorCallExpr 0x3798070 'class EdgeIterator' lvalue
20 | |-ImplicitCastExpr 0x3798058 'class EdgeIterator &(*)(void)' <FunctionToPointerDecay>
21 | | `-DeclRefExpr 0x3798000 'class EdgeIterator &(void)' lvalue CXXMethod 0x3795150 'operator
      ++' 'class EdgeIterator &(void)'
22 | `-DeclRefExpr 0x3797fd8 'class EdgeIterator' lvalue Var 0x3797d68 'eItr' 'class
      EdgeIterator'
23 `-CompoundStmt 0x37982b0
24   `-DeclStmt 0x37981a8
25     `-VarDecl 0x37980c0  used nvd 'struct VertexData &' cinit
26       `-CXXMemberCallExpr 0x3798180 'struct VertexData' lvalue
27         `-MemberExpr 0x3798148 '<bound member function type>' .getNeighVertexData 0x3794e90
28           `-DeclRefExpr 0x3798120 'class EdgeIterator' lvalue Var 0x3797d68 'eItr' 'class
                EdgeIterator'
```

Figure 4.3: Abstract Syntax Tree (AST) for a Gather Loop.

Static Analyzer which finds bugs automatically [11]. For our specific needs, Clang provides much more flexible and understandable Abstract Syntax Tree (AST) compared to other alternatives such as GCC [50].

Our main purpose in using Clang is to extract necessary data about the user code in order to translate it into SystemC. Therefore, AST is a very important feature since it holds the code structure with many details in it. For example, one can manually create completely different language version of any C-Language program using AST. Figure 4.3 shows the printable version of an AST subtree, which constructs the Gather loop in a graph application.

To extract the meta-data, we traverse each node in AST. Clang provides a visitor, named as RecursiveASTVisitor. It does pre-order or post-order depth-first traversal on an entire AST and visits each node. It also provides specialized traversal on some types of nodes such as Stmt and Decl. Table 4.1 lists the most frequently used data-types and functions in our implementation.

Therefore, using the meta-data, one can understand the AST structure and search

Types and Fuctions	Description
Decl	Any C++ declaration. e.g., VarDecl -> int accum; FunctionDecl -> void VertexProgram(vtx)
Stmt	Any C++ statement. e.g., IfStmt ->if (accum < vd.dist) ForStmt -> GA_FOREACH_EDGE(vtx,eItr,IN_EDGE) CompoundStmt -> {ovid++; vtx.getOtherVertexData(ovid);}
Expr	Sub-class of Stmt. e.g., MemberExpr -> vd.dist CallExpr -> vtx.getVertexData(), etc.
TraverseDecl(...)	Traverse all the declarations in the AST. e.g, to find the LVD fields and accumulation variables
TraverseStmt(...)	Traverse all the statements in the AST. e.g, the LVD field usage (read-only or read-write)

Table 4.1: Most frequently used Clang data types and functions.

the tree to extract data. For example, Figure 4.3 shows that, the root of this subtree is a ForStmt (Line 2) with a VarDecl and a type of EdgeIterator (Line 4). The declared variable is initialized to *vtx.beginEdgeIterator()* (Line 8). Rest of the subtree shows the limit of the iteration (Line 14), increment (Line 19), and the CompoundStmt (Line 23) executing the given Stmt for every iteration of ForStmt. This AST shows the structural information of gather loop of the BFS algorithm. This traversal only extracts the structural information which is not sufficient for our goals.

Figure 4.4 shows a Clang plugin implemented to find the referenced LocalVertex-Data fields and its usage (read-only or read-write). Moreover, if one of these fields is referenced inside a gather loop, where a vertex uses its neighbors' data, the field is also shared. That information will be used while creating the local tables and the data structure of *VertexDataShared* and *VertexDataPrivate*. We decided to divide *VertexData* in such structures, because, when a neighbor requests the data, only the required portion will be read from the memory subsystem.

Next step is the creation of the threads and the FIFOs. Since we aim to handle all types of graph applications, we cannot directly create a complete structure. Therefore, using gather loops, we divide the user code into smaller pieces. All

```
1
2  void recordUsageOfVertexDataFieldsUpstream(
3   MemberExpr *&foundExpr,
4   SimpleVariable &sv
5   ) {
6    foundExpr = NULL;
7    for (int si = nodeStack.size() - 1; si >= 0; --si){
8     if (Stmt *stmt = nodeStack[si].getStmt()) {
9
10     IF_DYN_TYPE(stmt, MemberExpr, mexpr) {
11      if (DeclAccessPair dap = mexpr->getFoundDecl()){
12       const CXXRecordDecl *rd =
13        mexpr->getBase()->getBestDynamicClassType();
14       if (rd && rd->getName() == "VertexData"){
15
16        referencedLVDFieldsNodeIndexes.
17         push_back(simpleNodes.size() - 2);
18        referencedLVDFieldsNodeIndexesVertexProgram.
19         push_back(inWhichVertexProgram);
20        bool isMutable =
21         searchModifyingOperatorUpstream(si);
22
23        foundExpr = mexpr;
24        sv = SimpleVariable(dap->getName(),
25         mexpr->getType(), isMutable);
26        addFieldToSet(sv, referencedLVDFields);
27        break;
28      }
29     }
30    }
31   }
32  }
33 }
```

Figure 4.4: Example method for finding referenced Local Vertex Data fields.

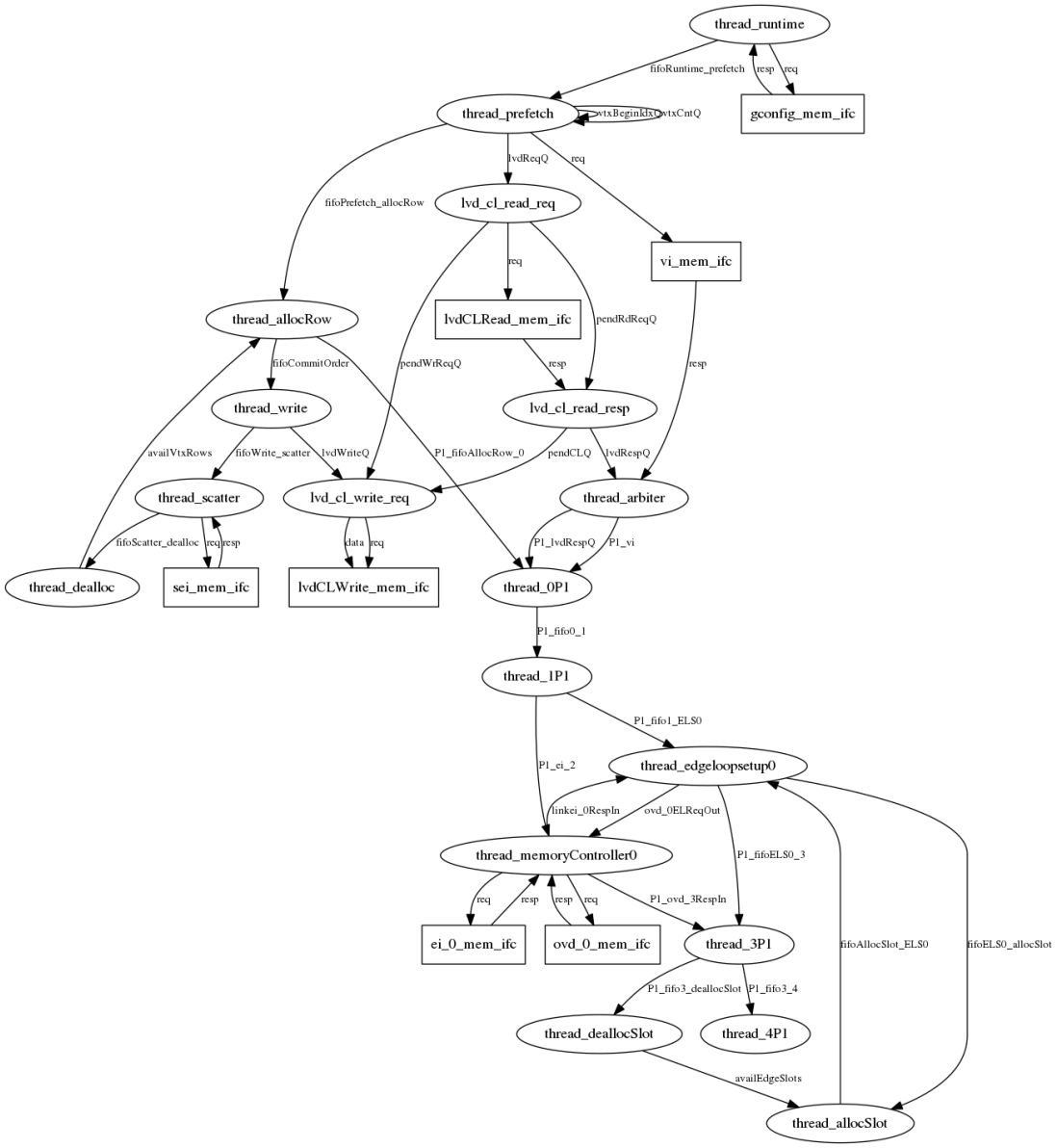

Figure 4.5: Thread communication graph for Breadth-First Search application.

these pieces can be executed asynchronously when necessary data arrives from the predecessor thread. In our case, most of the time, *rowId* is used to find the data in global tables, which will be discussed further in Sec. 4.2.3. FIFO structures are created since there needs to be communication among the threads. If there is no conditional gather loop, the structure is simple for the user code. Every thread receives data from the previous one and sends data to the next. On the other hand, if there is a conditional gather loop, then the FIFO structure will not be sequential. Figure 4.5 shows the threads and the FIFOs with the memory subsystem interface for a simple application like BFS.

As can be seen in Figure 4.6, all required information is extracted and collected from the AST. The communication with the memory subsystem, multiple vertex programs, neighbor activation are all necessary to generate the SystemC code. In our design, these are written in temporary object files.

After the execution of Clang tool, the next step is to create the SystemC files and connections between the threads, to insert program code into thread files, to specify data structures, and to generate arbiters to reuse the modules and the ports of the memory subsystem. After binding the threads to the memory subsystem ports, the SystemC model is ready to be tested and used.

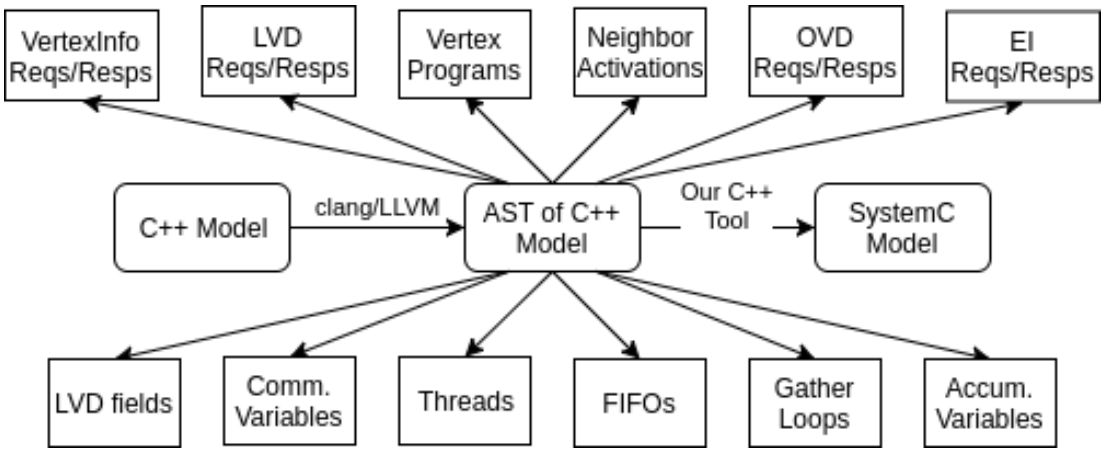

Figure 4.6: Collected meta-data from Abstract Syntax Tree (AST).

4.2.2 Fixed-Point

Fixed-point representation is a number format, which holds the number in two separate parts: the digits before and after the decimal point. These parts correspond to the fractional part and integer part of a number. The term "fixed-point" indicates that there is a fixed number of digits before or after the decimal point.

In the floating point representation, there is no such thing like fixed numbers of digits before or after the decimal point. The "float" refers to the decimal point which can be anywhere in the number and the place of the decimal point is regarded as the exponent, so floating-point representation is similar to scientific notation.

Therefore, using the same number of digits we can represent a wider range of numbers with the floating point since the place of the decimal point can make the number both large and small. However, floating point representation approximates the numbers to their real values because they cannot be exactly expressed and the gaps between the adjacent numbers vary which results in rounding a number to the nearest. Therefore, there is a trade-off between range and precision of the numbers.

Moreover, floating-point operations make the design more complex and the area required bigger [51] [52]. In general, FPGAs do not have floating point units, whereas, an efficient implementation of a fixed-point unit exists for some of them. Moreover, fixed-point is often used in hardware implementations because of its cost-effectiveness, smaller memory requirement, and narrow bus [52]. Furthermore, for the currently produced FPGAs that support floating-point usage, it is highly recommended switching to fixed-point for better performance [53][54].

Because of all these reasons, during source-to-source transformation, all the *float* typed-variables are translated into fixed-point. By doing this, we ensure that, the resulting source-to-source implementation and the latter stages of the framework will be supported by the target hardware without any inefficiencies due to the data type choices.

4.2.3 Global Tables

Since *VertexProgram* is mapped into a pipeline structure, a vertex follows all the modules in that structure if there is not any conditional flow. An example pipeline is shown in Figure 4.2. The application starts with a *VertexInfo* request, from the vertex view. This data structure holds the edge information for its neighbors, augmented data, etc. When memory responds with these data, they are stored in the global tables, which can be accessed for both reading and writing by all the modules. Therefore, using neighbor edge information in the *VertexInfo*, neighbor *VertexData* is also requested. Potentially, there are many neighbors for each node which will only be used once. Therefore, they are not saved on the tables, and disposed after use. When the accumulation finishes, the vertex data in the tables are updated by using both accumulation value and the variables used across the application. These values are also held in global tables since they can be used in different places. Data stored for a vertex is valid in the tables until there is an update in which case it requires an invalidation. The basic global tables are shown in Figure 4.2.

Due to the aforementioned reasons, the required data is kept in a special data structure throughout the lifetime of a vertex in the pipeline. This way data will not be requested from the memory every time.

4.2.4 Component-Based Template

In this section of the thesis, we will describe the template-based design methodology. As explained in the literature [2], a template-based hardware accelerator can potentially hide the latency emerging due to irregular communications, random DRAM accesses, and limited data locality, etc. Moreover, the template can be utilized for different graph applications, thereby eliminating the long design and testing processes in RTL [2].

Template-based design [2], requires user to specify necessary methods to build

the graph application, namely, *gather* and *apply*. The user also specifies the data structures that are used across the application. Therefore, using these specified methods, data structures, and the underlying template design, SystemC-based graph application can be created, with minimal effort. Thus, the user has the flexibility about applications and freedom of not knowing and facing low-level details such as message passing, synchronization, and parallelization.

As can be seen from Figure 4.2, Edge Loop Execution (ELExec) and Apply modules correspond to user created *gather* and *apply* methods, respectively. The other modules are part of the template-design used for creating the SystemC model.

In this work, we extend the accelerator flow described in [2], by adding a source-to-source transformation phase to provide more flexibility in types of accelerators that can be designed and to make it much easier to use the design framework from a user's perspective. This way user can write any graph application without being limited to GAS (see Sec. 4.2.1 for more details) and from that application the SystemC model can be generated.

As mentioned previously, there are some helper modules in the final pipeline structure used to support the template such as *AllocRow*, *Prefetch*, *InitVertex*, etc. These modules are shown in Figure 4.2, where they are not relevant for the user defined application details, nor the methods used for accelerator design flow in the previous work [2]. Rather, these modules are automatically generated modules to communicate with the memory interface and read from or write into the global tables.

Similar to the previous work [2], these modules are automatically added to pipeline structure. However, since we support multiple pipeline (see Sec. 6.6 for further details), some modules will be reused, such as WriteData, Scatter, Edgeloop Setup (ELS). These modules perform common tasks needed by different pipelines, such as requesting the neighbor data from memory, etc.

Since these modules are part of the template design and same for each application,

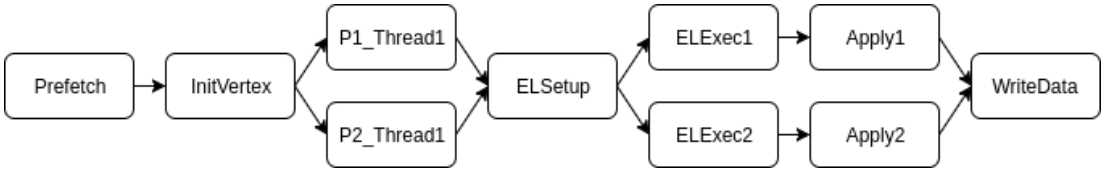

Figure 4.7: Basic architecture of a SystemC application with reused modules.

replicating them will be inefficient, in terms of complexity and area required in the FPGA. Therefore, we do not regenerate those modules and make the design as efficient as possible.

The best way of supporting multiple pipelines is not generating and duplicating the whole pipeline multiple times, but generating the template modules only once and reuse across other pipelines. Remaining modules like *ELExec*, *Apply*, which can vary for different *VertexPrograms* will be used specifically for that pipeline. Note that, there is a module named *Thread1* with a prefix demonstrating the pipeline. The aim of generating such module is to initialize the local variables and handle the conditionals across the pipeline structure.

Figure 4.7 shows a small and simplified example of reusing the modules across the application. Note that, *Prefetch*, *InitVertex*, *ELSetup*, *WriteData* are reused for multiple pipelines. Therefore, we can support multiple pipelines without duplicating all of the modules to make the application space smaller and more efficient.

4.2.5 Vertex Program

One critical component in a graph application provided by the user is *VertexProgram*. This is the part of the application that will be converted into pipeline structure. Therefore, there are some rules and limitations to be considered.

First of all, the application is specified with the *VertexProgram* keyword. In case if there are multiple VertexPrograms, the naming should be *VertexProgramN* where "N" denotes the number of the "VertexProgram".

Additionally, user needs to specify the data structure of the *VertexData* including members held by the *VertexData* object.

Reading *VertexData* is handled by *VertexHandle* object. This is a shim object and is just used to make the application generic. Since this is an application modelled in "Think-Like-A-Vertex" [17] model, the vertex reads its data using *getVertexData()* function, which can be seen in Figure 4.8 Line 7.

For the gather part, user has to write a "for loop" over the *INEDGES* or *OUT-EDGES*. This is the part where vertex gathers the data from its neighbors and accumulates them into a variable. Line 10 in Figure 4.8 shows the details of a for loop that iterates over the *INEDGES*.

Accumulating the neighbor data into a variable takes place in the for loop mentioned above. To do that, first, a vertex must request the neighbor data using *EdgeIterator*. After the arrival requested data, the vertex can use the neighbor data. Line 11 in Figure 4.8 shows the requests from the neighbors. Also, a vertex can decide to activate the neighbor being processed, which is mentioned in Section 6.3.

At the end of the vertex program, accumulated data is used to update the local data fields of the corresponding vertex.

```
1
2  void VertexProgram1(VertexHandle &vtx) {
3
4    double accumMin;
5    accumMin = 10000000;
6
7    VertexData &vd = vtx.getVertexData();
8
9
10     GA_FOREACH_EDGE(vtx, eItr, IN_EDGE) {
11       VertexData &nvd = eItr.getNeighVertexData();
12       double newDist = nvd.dist + 1;
13
14       if(newDist < accumMin){
15         accumMin = newDist;
16       }
17     }
18
19
20     if (accumMin < vd.dist){
21       vd.dist = accumMin;
22     }
23 }
```

Figure 4.8: Simple user code given as an input to our accelerator design flow. Breadth First Search (BFS) algorithm functions are written in C++.

Chapter 5

Source to Source Transformation

Program improvement can be achieved in many ways, one of which is to use source-to-source transformation. Using a compiler or a code analysis tool, a program written in a certain language can be optimized for the same language or partly/completely translated into a different target language. However, improvements can be beyond optimization and efficiency, such as enhancing the user experience. Although hardware accelerators increase the efficiency in orders of magnitude, the underlying implementation for these architectures such as GPU, FPGA, and ASIC, are much harder when compared to a high level language such as C++. Therefore, if a developer cannot use the accelerators due to cost of hardware and adversities of the implementation, he or she will have to accept the outcome of CPU execution. On the other hand, the existence of a tool that accepts a simple C++ code and can execute on an FPGA will provide benefits of both worlds and will allow executing on larger graphs.

As mentioned before, our ultimate aim is creating hardware accelerator framework using source-to-source transformation to handle wider range of applications than what has been previously proposed in the literature [2]. Therefore, in order to provide both ease of development and flexibility in expressiveness, we have added the source-to-source transformation on top of our model. With this, user can write any kind of graph application in C++ with her/his desired data structure

for vertices, and the application is translated into SystemC regardless of the number of gather loops, the number of vertex programs, conditional memory requests, or the neighbor activation required, etc.

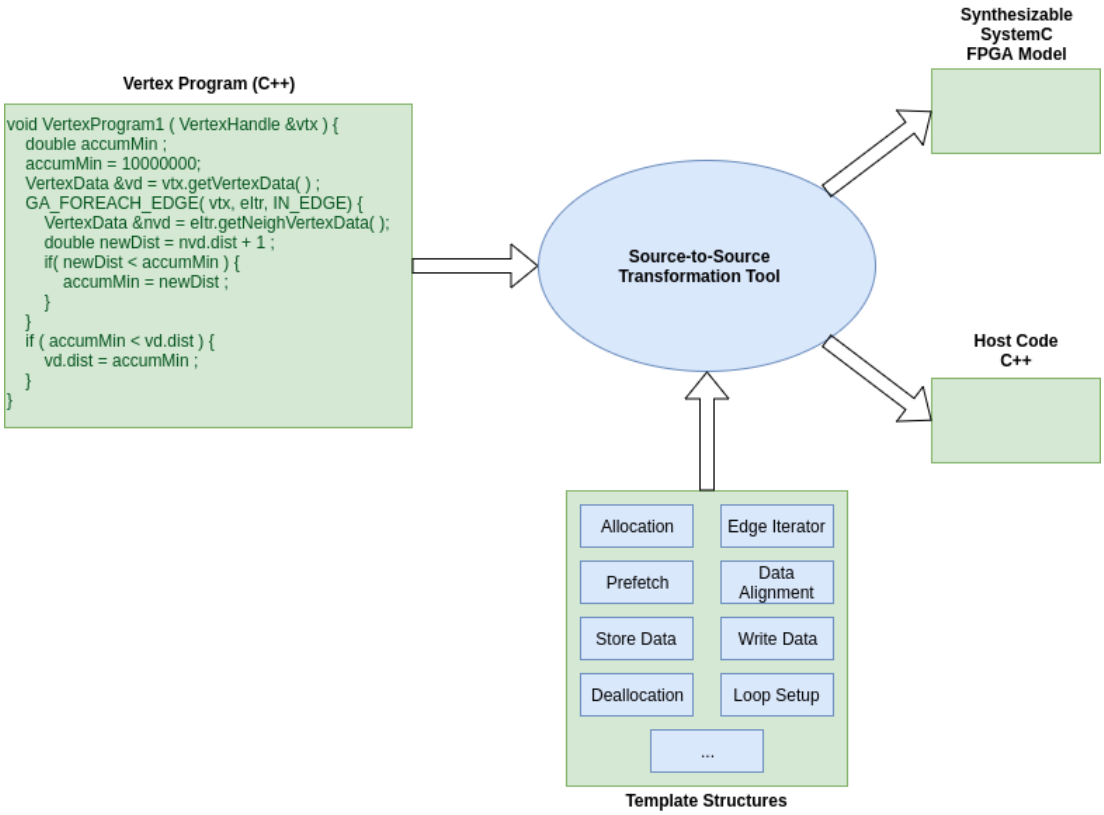

Figure 5.1: High level view of source-to-source transformation step

Figure 4.8 illustrates the basic graph application given in C++. As can be seen, the application starts with local variable declaration, initialization and *VertexData* object creation. Then, gather loop execution takes place in which neighbor data is obtained and used to accumulate. This is followed by a state change for the currently-processed vertex. While in this simple example, Breadth First Search (BFS) algorithm within the GAS model is implemented, it is possible to support non-legacy applications, more complex structures that use multiple vertex programs, multiple gather loops, conditional gather loops, etc. These additional features will be discussed in detail.

As can be seen in Figure 5.1, our Source-to-Source Transformation tool takes a Vertex Program from the user along with the template structures [2]. Using

these inputs, tool generates a synthesizable SystemC FPGA model and a C++ host code. During the latter parts of the accelerator flow, SystemC model is synthesized and used to generate the FPGA programming image and the host code is used to run the FPGA model.

Chapter 6

Accelerator Generation

Accelerator generation involves different features including asynchronous execution, supporting non-GAS applications, active set, conditional iteration over the neighbors, enabling non-neighbor data access, multiple vertex program support and user-defined data types. These features will be highlighted in the following sections.

6.1 Asynchronous Execution Support

Our proposed architecture establishes asynchronous execution on iterative graph-parallel applications. Vertices gather the data from neighbors, calculate the accumulated data, change its state and write to the global table data structure (see Sec. 4.2.3). Then, neighbors can access the data that is written on the table which is the most recent calculated data. So, vertices are not forced to accumulate using neighbor data from the previous iteration.

6.2 Non-GAS Application Support

The main limitation in the GAS model is that there can only be one from each stage. Therefore, user can only specify gather, apply, and scatter stage methods only once. However, there can be applications in which user may want to gather the incoming-edge neighbors and accumulate their data using a specific function first, followed by a gather function on the outgoing edge neighbors and accumulate their data using another function. Between neighbor iterations and at the end of iterations, the user may want to write the collected data to its local state using an apply function. In such circumstances, GAS abstraction does not fulfill the needs of the user. On the other hand, our application interface gives user the flexibility to implement applications beyond GAS model. More specifically, user can develop not only GAS applications which can only include one for each function described above, but also non-GAS applications with possibly multiple gather, sum, apply, and scatter stages.

Algorithm 1: Non-Gas Application Support

```
1 foreach n in v.inNeighbors do
2 |   accum1 += v.field1
3 end
4 foreach n in v.inNeighbors do
5 |   accum2 += accum1 + v.field2
6 end
7 foreach n in v.inNeighbors do
8 |   accum3 += accum2 * v.field1 + accum1 * v.field2
9 end
```

Algorithm 1 shows the usage of non-GAS application support in pseudo-code. Note that, there are multiple gather loops and between those, user can use the collected data. Each gather loop may accumulate the data on a different variable and use the data from another loop.

6.3 Active Set Support

For a graph application, the number of iterations needed to converge for a vertex can vary dramatically. The study on iteration number for the vertices on PageRank shows that, 7.4% of the vertices converge in a single iteration, 51% of the vertices converge in 36 iterations, whereas 99.7% converge in 50 iterations. For the remaining part, 0.3% of the vertices require 27 more iterations [44]. The same study shows that total work-done by processing only active vertices during different iterations reduces the total computation by nearly 50% [44]. This study clearly states that it is inefficient to process all the vertices in every iteration of a graph application.

For example, Graph-Coloring is a vertex labeling problem in which two adjacent vertices will not be assigned to the same color. Maximal Independent Set (MIS) is used to solve this problem [55]. In every iteration, MIS is constructed and assigned a new color, then a vertex is removed from the graph, until there is no vertex left in the graph. Moreover, MIS algorithm [56] is used by Luby's classic parallel algorithm which uses conditional activation of the neighbors. The algorithm starts with assigning random values to all the vertices and for each vertex, checks whether the value for the vertex is the smallest among itself and its neighbors. If so, removes the vertex from the graph and puts it into an "independent set". If this is not the case, activates all the neighbors.

Algorithm 2: Active Set Support

```
1 foreach n in v.inNeighbors do
2 |   if some condition then
3 |   |   activateNeighbor(n)
4 |   end
5 end
6 if some condition then
7 |   v.activateNeighbors()
8 end
```

In our implementation, a vertex can have two different states: active or inactive. At the beginning, all vertices are considered to be in the active set. During the

computation, only vertices in the active set participate in the execution of the *VertexProgram*. Once the iteration finishes, vertices are removed from the active set, that is, a vertex will not participate in the calculation anymore unless it is triggered (activated) by its neighbors. Therefore, during the execution, vertices can activate their neighbors, causing these neighbors to be added to the active set and processed during the next iteration. Vertex program will continue until there are no active vertices or the maximum iteration number is reached.

Our application interface gives user the flexibility about the activation of neighbors as well. The developer does not have to activate all the neighbors of the currently processed vertex. Instead, a set of neighbors can be placed into the active set. As seen in Algorithm 2, while iterating over the neighbors, the user can select the neighbors according to one or more conditions. If needed, it is also possible to allow a vertex to activate all the neighbors at once.

6.4 Conditional Pipeline Support

As explained before, Maximal Independent Set (MIS) is used in the implementation of Graph Coloring, where Color2MIS [57] works for a given vertex coloring scheme. In this application, edge iteration over the neighbors of a vertex takes place if the coloring of the vertex is equal to the currently processed color. Therefore, the corresponding iteration will take place conditionally.

Similar to the above application, developer may want to implement edge-iteration depending on a single condition or even multiple conditions. Our accelerator development flow handles this case and converts such an application to a conditional pipeline. When the conditional check takes place in this pipeline, data (rowId) is placed into different FIFOs and sent to respective modules.

Algorithm 3 illustrates how the conditional pipeline design works. As clearly seen, if the first condition is met, edge iteration over the neighbors takes place. Note that, there is an additional condition when the edge iteration is not executed.

Algorithm 3: Conditional Pipeline Support

```
1  if some condition then
2  |  foreach n in v.inNeighbors do
3  |  |  statement
4  |  end
5  else
6  |  if some condition then
7  |  |  statement
8  |  else
9  |  |  foreach n in v.inNeighbors do
10 |  |  |  statement
11 |  |  end
12 |  end
13 end
```

6.5 Non-Neighbor Data Access

To describe what we mean by non-neighbor data access, we will use Shiloach-Vishkin algorithm [58] which aims to find connected components in a graph structure. In this application, every vertex holds the id of non-neighbor vertices, to compare its data with the non-neighbor vertex data. Therefore, the need for accessing non-neighbor data is required for this problem.

Consider a similar application in which every vertex in the graph has an id of another vertex, namely *ovid*, and searches for the neighbor which has the minimum label value. Moreover, each vertex desire to access the vertex data with *id* = *ovid*, where *ovid* is held by the neighbor vertex with minimum label.

In such a scenario, using a basic edge iterator loop over the *InEdges*, one can find the minimum label among the neighbor vertices and accordingly update the *ovid* variable. Once edge iteration over neighbors finishes, *VertexData* of the vertex with the *id* = *ovid* can be gathered. Algorithm 4 shows the usage of non-neighbor data access.

Algorithm 4: Non-Neighbor Data Access Support

```
1 VertexData& ovd = vtx.getOtherVertexData(ovid);
```

This is not supported in a standard GAS application framework, thus, we provide user a wider range of application development opportunities.

6.6 Multiple Pipeline Support

Hyperlink-Induced Topic Search (HITS) [59], or with the well-known name Hubs and Authorities algorithm, is a ranking algorithm like PageRank [1]. However, HITS works with not only in-links but also out-links of a vertex, and assigns each vertex different scores, namely, hub and authority. The authority score is higher if vertices are pointed by many links, otherwise, the hub score is higher. First, using hub scores, authority score of all vertices are calculated and normalized. Then, using authority scores, hub scores are calculated.

Algorithm 5: Multiple Pipeline Support

```
1  Function VertexProgram1(vtx)
2      foreach n in v.inNeighbors do
3          statement
4      end
5  end
6  Function VertexProgram2(vtx)
7      foreach n in v.outNeighbors do
8          statement
9      end
10 end
11 Function main()
12     foreach v in V do
13         VertexProgram1(v.vertexHandle)
14     end
15     foreach v in V do
16         VertexProgram2(v.vertexHandle)
17     end
18 end
```

Algorithms like HITS requires not only iterating over the edges multiple times but also over the vertices. In our programming interface, as mentioned above, for a given subset of vertices, the *VertexProgram* is executed. Moreover, every vertex

program is converted into a pipeline structure, on which data (*rowId*) flows. So, if the user implements an application with multiple *VertexProgram*, the whole application is converted into a multi-pipeline structure. Algorithm 5 shows the usage and interface of multiple pipelines or multiple vertex programs.

6.7 User-Defined Types

Fixed-point is basically a number representation using fixed digits before and after the decimal point. As mentioned in section 4.2.2, during the SystemC code generation, all *float* type variables are converted into fixed-point. The default fixed-point consists of 16 bits of the integer part and 16 bits of the fractional part.

Algorithm 6: User-Defined Types Support

```
1 typedef udt<24,1> udt1;
2 typedef udt<48,16> udt2;
3 udt2 variableName;
```

Therefore, this version of the fixed-point representation may not fulfill the needs of the user, such as working with the positive numbers less than 1 which only needs a fractional part, etc. For such circumstances, we propose user-defined types. With the help of this feature, one can define a type with the desired parameters. Algorithm 6 shows the definition and usage of the types. The first parameter of the defined type is the total bit count, whereas the second parameter shows the place of the decimal point. *udt* stands for user-defined type and it is a shim data structure which is created to make type definitions generic. As it can be seen, *udt1* type has 24 bits, 23 bits of which represents fractional part, whereas *udt2* type has total of 48 bits with 32 bits of fractional part. After the definition, the new type can be used to create variables across the application.

Chapter 7

Experimental Evaluation

In this section, we present our experimental evaluation. We first give the detailed explanation of the platforms and architectures used to implement the flow. Then, we describe our graph parallel benchmarks and their respective properties. We also provide different implementation versions which we compare. Lastly, we give our experimental results with different settings.

7.1 Experimental Setup

7.1.1 Clang

Clang is the most important part of the flow, used for Source to Source Transformation (S2S). It is a front-end tool provided by LLVM Project [49] and it is a compiler which has an active community and growing industry support [11]. It can be used to parse C/C++ code, in our case, the user code. To translate the user code into SystemC, we implemented our Clang parser plugin and transform C++ source code into an Abstract Syntax Tree (AST) from which we extracted the information that is used in SystemC like vertex data members such as read-only data, read-write data, private data, shared data, edge-iterator loops.

7.1.2 SystemC

SystemC [60] is a hardware description language using a set of C++ classes. It provides modeling and event-driven simulation interface for the developers designing hardware systems. This language is similar to Hardware Description Languages (HDL) like VHDL and Verilog in the aspects of designing the modules and connections between, however SystemC is system-level modeling language in contrast to HDLs which are register-level modeling.

SystemC is used with transaction-level abstraction, generally. Transaction-level modeling (TLM) is the high-level abstraction of digital systems which allows developers and designers not to concern about the low-level details of the communication between the modules, instead, the focus is mostly on functionality, performance, and verification of the communication [61].

In the latter parts of the design stage, transaction level modules are converted into Register Transfer Level (RTL) design which can be any HDL like Verilog or SystemVerilog, to analyze the performance in detail and verify the system.

7.1.3 CtoS

Cadence's Stratus C-to-Silicone compiler [46] is a High-Level Synthesis(HLS) tool which aims to create and design efficient and optimized hardware abstraction of the given software which is algorithmic description in a high-level language such as SystemC/C [62] with the module functionality and inter-module communication. Briefly, CtoS translates algorithmic behavior to register transfer level. HLS consists of a number of activities such as algorithm optimization, scheduling, register binding, etc. [62, 63]. After applying these series of algorithms on given software, Register Transfer Level code (RTL) is generated which is design abstraction modelling the digital circuit with the data flow between the hardware registers [62].

7.1.4 Intel Accelerator Simulation Environment

The Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) is a hardware-software co-simulation environment [64]. The environment grants the developers such features like fake physical memory, FPGA local memory model, inter-process communicator that the developers simulate the FPGA board. Such simulation reduces the development time for AFU hardware and software by providing both functional validation and removing design problems like lock conditions, data dependency hazards, pointer math errors and address mapping.

ASE provides 2 interfaces: the hardware side and the software side. Hardware interface basically communicates with the RTL which is generated by CtoS and mimics FPGA. On the other hand, software interface communicates with the host code [64].

The main aim of the ASE simulation is to see whether there are some design errors from the development or generation stages and solve those problems before generating bitstream which takes a very long time.

7.1.5 Intel Quartus Prime

Quartus is the last step of the design flow. It is a logic synthesis tool like Xilinx ISE [65] and is used to synthesize HDL and RTL designs, to analyze timing performance, to inspect and simulate the designs and to generate the bitstream.

After successful completion of ASE, the generated RTL is considered most likely to be correct. Therefore RTL code can be used to generate bitstream files. Bitstream generation consists of several steps like synthesis, device and resource assignments, place and route by fitting the logic of design into the device, selection of interconnection paths, pins and logic cell assignments. The last step is assembler which generates a device programming image, in our case, bitstream. Eventually, the image can be loaded and executed on the FPGA. [48]

7.1.6 Hardware Specifications

We have used heterogeneous high-performance computing workstations consisting of four 22-core Intel Xeon E5-2699 CPUs with the base frequency of 2.20 GHz and Arria 10 GX1150 FPGA to evaluate our application and OpenCL implementations. These workstations are provided by Hardware Accelerator Research Program (HARP), which is mentioned in Section 3.5.

7.2 Graph Applications

Our flow aims to perform better in two different metrics, the performance and the work done. In order to support both objectives, we have implemented our flow in two different modes, namely, high-throughput (HT) and work-efficient (WE). Before giving details about these two modes, we first describe the legacy algorithms implemented for experimental evaluation in the following sections.

7.2.1 Breadth First Search (BFS)

Breadth First Search is a graph traversal algorithm, which starts exploring from a start node and assigns levels to the vertices. This algorithm is suitable for both high-throughput and work-efficient models, since visited vertices can be excluded from execution. HT version is straightforward, whereas in WE version, each vertex has a flag showing whether the vertex is visited or not. If the vertex is visited, then it will not be processed anymore, thereby working only on the active vertices.

7.2.2 Pagerank (PR)

Pagerank is a well-known ranking algorithm for graph structures. This algorithm is suitable for both models, too. In WE version, there is a control mechanism that decides if the vertex converges to a rank. After the convergence, it will not be processed anymore, unless some other vertex activates it.

7.2.3 Maximal Independent Set (MIS)

Maximal Independent Set finds the maximal set of a graph in which there are no neighbor vertices. It is implemented in only HT execution model. In the WE model, vertices deactivate the neighbors when they are selected for MIS. Since we are not supporting deactivation right now, WE-MIS model cannot be efficiently implemented. Instead, we modeled the application such that vertices decide not to activate themselves using an additional field. However, reading an extra field from the memory reduces the performance, so we decided not to implement MIS for work-efficient model.

7.3 Implementation

We made experiments with three different versions of each application, which can be summarized as follows:

- OpenCL: This is the base version against which we compare all the schemes. This version is implemented in OpenCL for the chosen algorithms using high throughput pull-based execution model and task parallelism with 4 kernels. Moreover, the accelerator frequency values obtained for this model are approximately 210MHz, 190MHz, and 220 MHz for BFS, PR, and MIS, respectively. Applications are evaluated using different number of kernels and the best results are obtained from the accelerator with 4 kernels. In

this implementation, throughput values for all algorithms is 1 edge/cycle, namely, in every cycle of a kernel execution, a result for a vertex pair is calculated.

Moreover, OpenCL does not support a handy active-set, but it can be implemented using push-based implementation model. However, due to random nature of memory accesses over the neighboring vertices, the efficiency and throughput drops significantly making this an unrealistic scenario. For our experiments on push-based OpenCL implementation of WE-BFS algorithm, the maximum throughput value we can achieve is 40 million edges per second with approximately 180MHz accelerator frequency.

- HT: This represents an approach that is implemented in our flow with high-throughput (HT) focus. In this version, all the vertices in the graph structure are processed in every iteration of the graph application, that is, we apply the synchronous execution model.

- WE: This represents a work-efficient (WE) scheme, which uses an active-set to hold the vertices participating in the computation. The rest of the vertices will not be processed in this scenario, achieving asynchronous execution.

Our HT version evaluation results will be compared to Base implementation to show the performance of our approach. On the other hand, WE implementation will be compared to both our HT version and Base implementations, to show that there is unnecessary work-done for some specific graph applications.

7.4 Datasets

The generated graph structures for the evaluation purposes are Kronecker graphs with different sizes and average vertex degrees [66]. For HT, we managed to use larger graphs than WE, because the active-set can hold up to 8 millions of vertices

at most, due to the limitations of FPGA area. Therefore, smaller graphs are used to evaluate WE.

7.5 Experimental Results

7.5.1 High Throughput Execution

Besides being more accessible, flexible and user-friendly for an inexperienced developer, our HT performs slightly better than OpenCL counterpart in terms of throughput and runtime. Since it is ensured that OpenCL application is implemented extremely efficiently, exceeding its performance is a good metric for our application in terms of performance. Due to the low-level details of parallel execution design, the development process in OpenCL is much harder than basic C++. Moreover, our approach provides slightly better results in terms of efficiency and speed. Therefore, we can conclude that our application outperforms the OpenCL in vertex-centric graph applications.

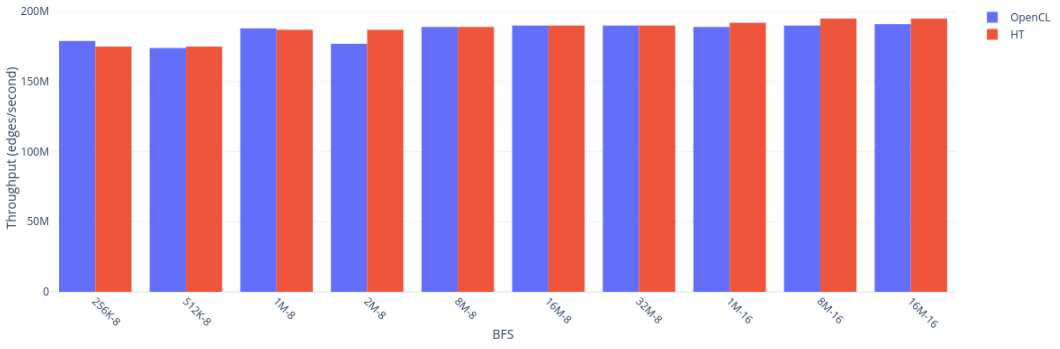

Figure 7.1: Comparison of OpenCL and HT in terms of throughput for BFS.

Figures 7.1-7.6 compare OpenCL with HT for BFS, PR, and MIS algorithms. Note that, results are given for different graph sizes with different average edges per vertex. Bar-charts show the throughput (edges per second) of the execution, whereas line-charts show the runtime of a single iteration (seconds). Since both

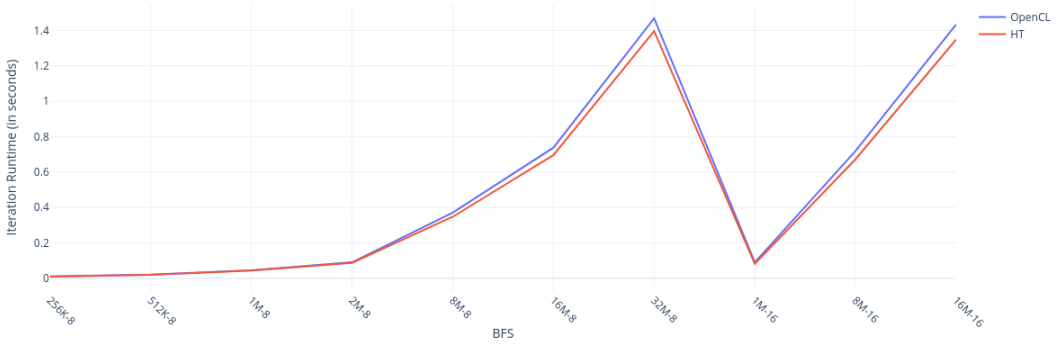

Figure 7.2: Comparison of OpenCL and HT in terms of a single iteration runtime for BFS.

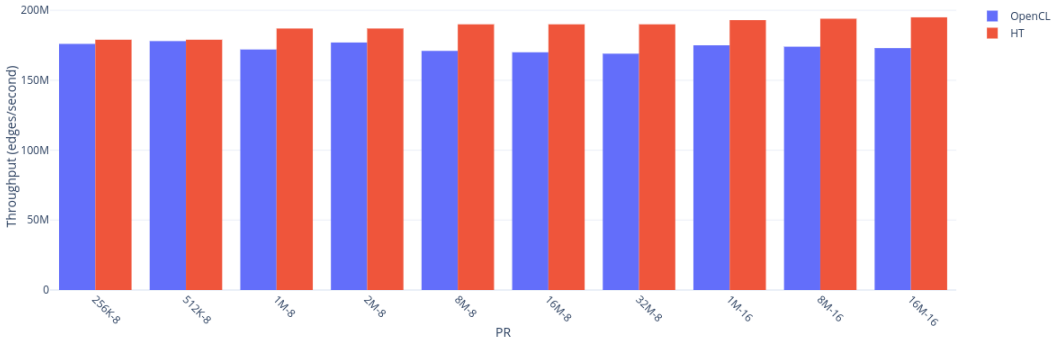

Figure 7.3: Comparison of OpenCL and HT in terms of throughput for PR.

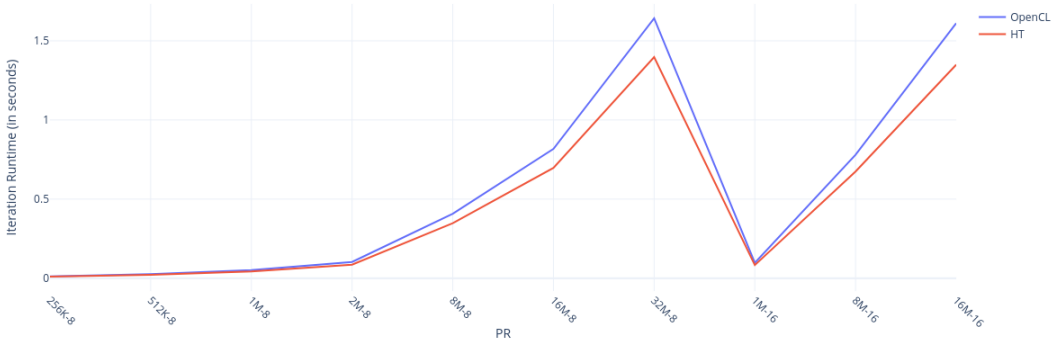

Figure 7.4: Comparison of OpenCL and HT in terms of a single iteration runtime for PR.

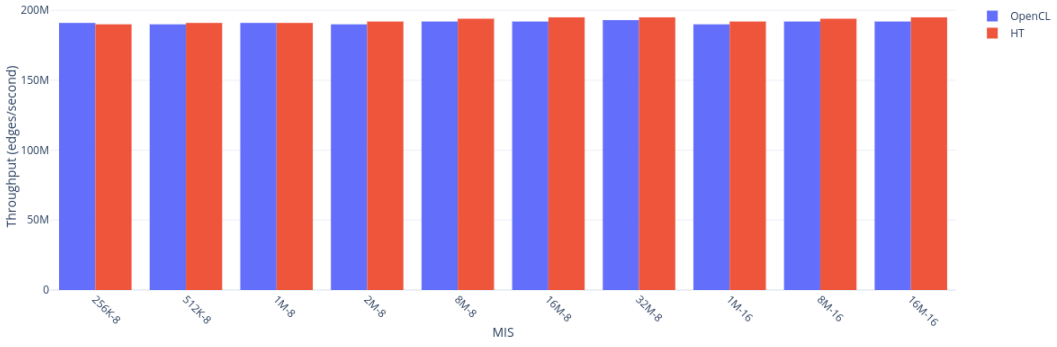

Figure 7.5: Comparison of OpenCL and HT in terms of throughput for MIS.

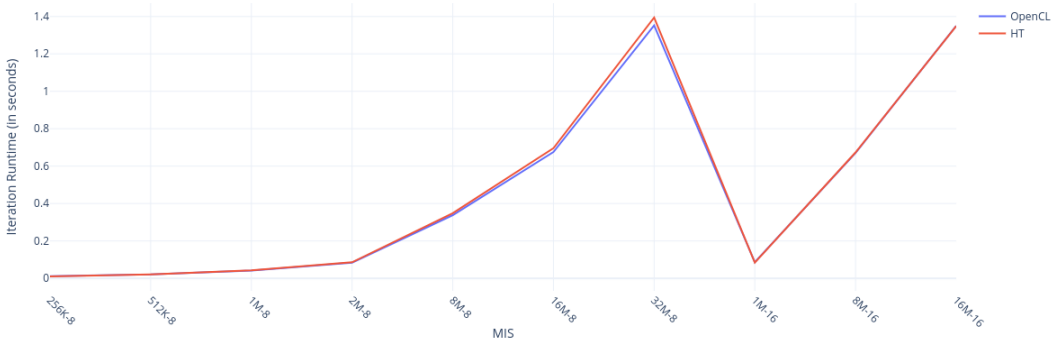

Figure 7.6: Comparison of OpenCL and HT in terms of a single iteration runtime for MIS.

implementations are evaluated using the same graph structure, the throughput values and iteration runtime of both HT and OpenCL implementations are proportional to each other. As stated, our model performs slightly better both in terms of throughput and execution time of a single iteration.

7.5.2 Work Efficient Execution

As stated in Section 6.3, different in and out degrees results in a varying number of iterations needed to converge for a vertex. Therefore, there may be unnecessary processing for converged vertices, which will result in with both time and energy cost. Since our accelerator development flow supports the active-set, work-efficient version does significantly less work than both OpenCL and HT counterparts. This also results in faster execution and convergence.

To fairly evaluate the models in terms of work done and runtime, we need to determine the number of iterations since HT and OpenCL models will be executed accordingly. We decided the maximum iteration numbers by running WE until all vertices converge. The number of iterations obtained from WE is given as an input to the other models, which ensures all the vertices will eventually converge in the maximum number of iterations in the worst case.

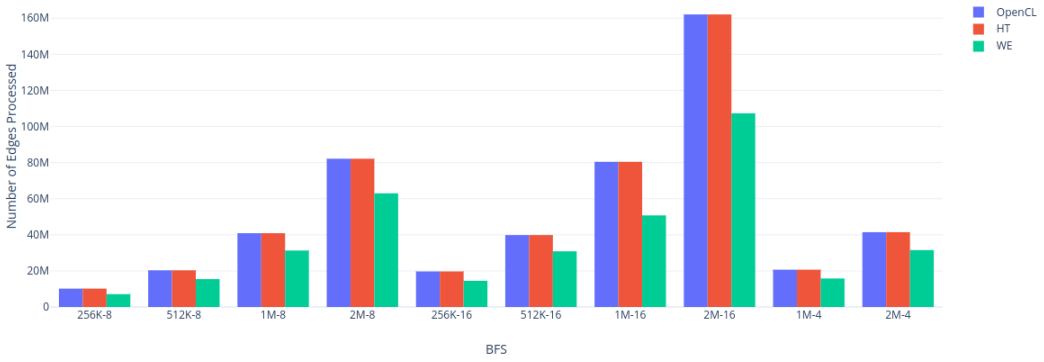

Figure 7.7: Comparison of OpenCL, HT and WE in terms of number of processed edges for BFS.

Figure 7.7 shows the comparison of 3 different models using the same graph structure and executing the BFS algorithm. Since HT and OpenCL counterparts do not support active-set, all the vertices will be processed during the computation until the maximum number of iterations. Unlike the other implementations, WE excludes the converged vertices from the active-set in order not to process them unnecessarily. On average, WE performs approximately 27% less computation for the BFS algorithm. Moreover, Figure 7.8 shows the same experiments' runtime. As can be seen, due to the lower computation, WE reduces the execution time by approximately 28% when compared to both HT and OpenCL implementations. This reduction also represents the savings in work done.

The WE performs better for the PR algorithm than BFS, for both metrics, i.e., number of processed edges and execution time until global convergence. The reason for this improvement is due to the fact that there are much more vertices converging earlier in the PR algorithm. Thus, a smaller set remains after a few iterations and that causes faster execution. Figure 7.9 compares the number of edges processed for different graph structures and vertex degrees. The reduction in work done is approximately 48%, on average. Figure 7.10 illustrates runtime of the PR algorithm for different execution scenarios with around 49% decrease.

Using work-efficient implementation, we can also compare throughput values

Figure 7.8: Comparison of OpenCL, HT and WE in terms of execution time for BFS.

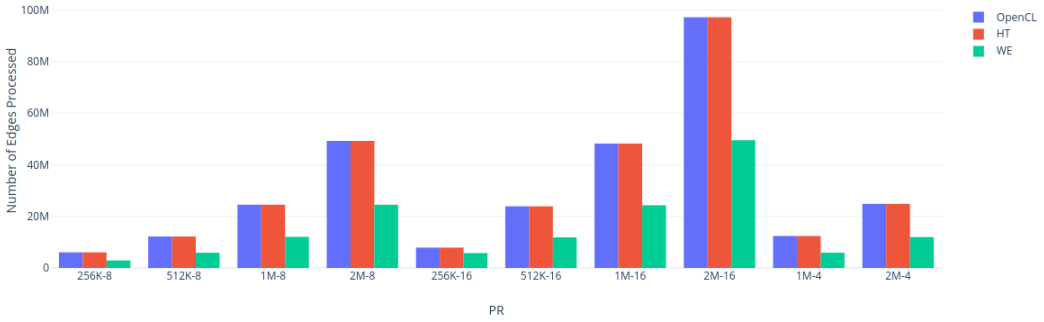

Figure 7.9: Comparison of OpenCL, HT and WE in terms of number of processed edges for PR.

Figure 7.10: Comparison of OpenCL, HT and WE in terms of execution time for PR.

with HT and OpenCL counterparts. Figure 7.11 and Figure 7.12 compares the throughput values of aforementioned models for different graphs, in terms of both size and average vertex degree. Note that, the throughput is nearly same for HT when compared to others, which shows that using an active-set and connecting it to the memory subsystem to access the neighbors does not slow down the application much. Moreover, for some average vertex degree values, WE version gives better results due to faster convergence.

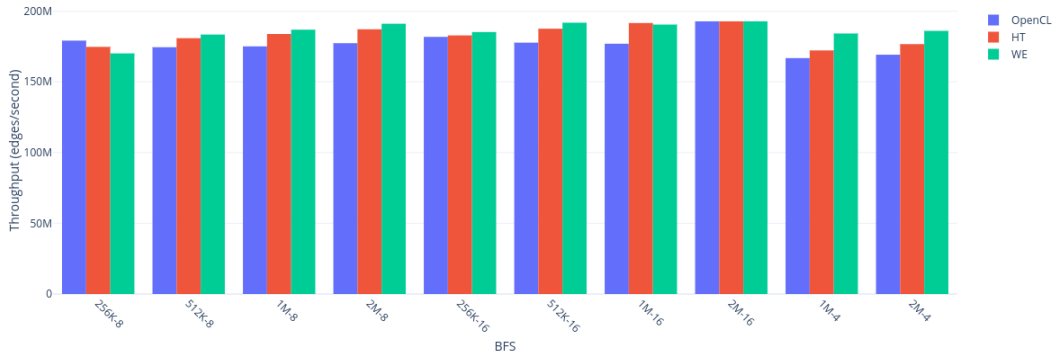

Figure 7.11: Comparison of OpenCL, HT, and WE in terms of throughput for BFS.

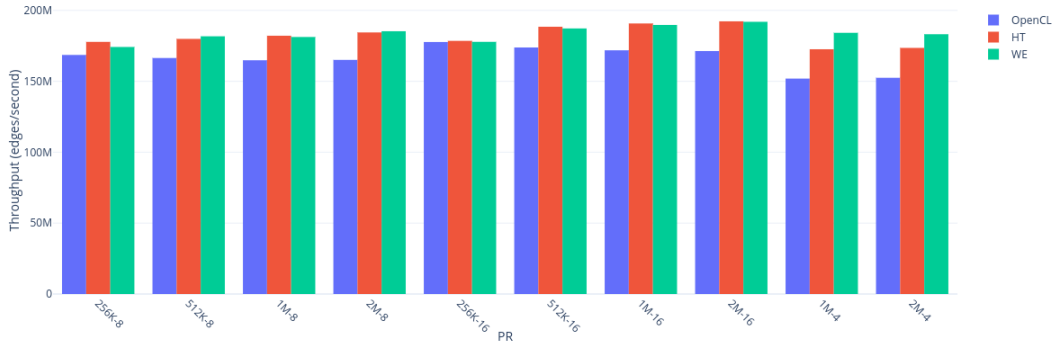

Figure 7.12: Comparison of OpenCL, HT, and WE in terms of throughput for PR.

The main metric showing the efficiency of the WE is the number of edges processed until the convergence and execution time is directly affected by the number

of edges processed. Besides, by having the same level of throughput, WE outperforms the OpenCL implementation, with approximately 40% less work done, averaging PR and BFS results.

The performance of WE is expected to be higher in terms of work done and execution time when used on real-world graphs, such as social, biological networks, etc. Because in those types of graphs, a higher portion of the vertices has lower degree, so they will converge faster and only the small set of high degree vertices will continue to be processed. However, HT will process those low-degree vertices also until the high-degree vertices converge.

Therefore, WE is more functional when there is a control mechanism to discard the vertices from the execution since it nearly performs as well as the HT. This way, it reduces the work done and more importantly, the execution time.

Chapter 8

Conclusion

The ultimate goal of this thesis is to discuss the implementation details of generating FPGA accelerators for graph parallel applications. This implementation uses a simple vertex-centric graph application representation given as a high-level language description. The main focus of the work presented here is extracting the necessary program information using abstract syntax tree (AST) and generating the intermediary objects from plain C++ code. Once the meta-data is collected, our architecture flow generates the SystemC model, which is followed by generating RTL and bitstream. Then the application can be directly executed on the FPGA without going through the low level development processes.

Therefore, the main contribution of this thesis is to create graph parallel hardware accelerators using source-to-source transformation. Any vertex-centric application can be modelled using our framework, can successfully be translated into SystemC, and can be executed on FPGA. There are various features in our accelerator design flow which provides a great level of flexibility to the user. These include asynchronous execution support, non-GAS application support, active set support, conditional pipeline support, non-neighbor data access support, multiple pipeline support and user-defined data types.

Based on our experimental results, by providing asynchronous execution with the

active-set, accelerator performance increased approximately 40% percent in terms of work done and total execution time when compared to its OpenCL counterpart for applications like BFS and PR.

In conclusion, we provide an easy FPGA accelerator framework for users without deeper knowledge of parallelization and optimization at the hardware level, but willing to increase the performance of their vertex-centric graph applications.

Bibliography

[1] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[2] Andrey Ayupov, Serif Yesil, Muhammet Mustafa Ozdal, Taemin Kim, Steven Burns, and Ozcan Ozturk. A template-based design methodology for graph-parallel hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(2):420–430, 2018.

[3] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with polaris. *Computer*, 29(12):78–82, December 1996.

[4] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, December 1994.

[5] Chunhua Liao, Daniel J. Quinlan, Jeremiah J. Willcock, and Thomas Panas. Extending automatic parallelization to optimize high-level abstractions for multicore. In Matthias S. Müller, Bronis R. de Supinski, and Barbara M. Chapman, editors, *Evolving OpenMP in an Age of Extreme Parallelism*, pages 28–41, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[6] Aart Bik, Milind Girkar, Paul Grey, and X. Tian. Efficient exploitation of parallelism on pentium iii and pentium 4 processor-based systems, 2001.

[7] Matthew Marangoni and Thomas Wischgoll. Paper: Togpu: Automatic source transformation from c++ to cuda using clang/llvm. *Electronic Imaging*, 2016(1):1–9, 2016.

[8] David Williams, Valeriu Codreanu, Po Yang, Baoquan Liu, Feng Dong, Burhan Yasar, Babak Mahdian, Alessandro Chiarini, Xia Zhao, and Jos BTM Roerdink. Evaluation of autoparallelization toolkits for commodity gpus. In *International Conference on Parallel Processing and Applied Mathematics*, pages 447–457. Springer, 2013.

[9] Siegfried Benkner. Vfc: The vienna fortran compiler. *Sci. Program.*, 7(1):67–81, January 1999.

[10] Edward B Duffy, Brian A Malloy, and Stephen Schaub. Exploiting the clang ast for analysis of c++ applications. In *Proceedings of the 52nd annual ACM southeast conference*, 2014.

[11] Clang: a C language family frontend for LLVM. `http://clang.llvm.org/`. [Online; accessed 19-June-2019].

[12] Tim Schmidt, Guantao Liu, and Rainer Dömer. Automatic generation of thread communication graphs from systemc source code. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, pages 108–115. ACM, 2016.

[13] Kevin Marquet and Matthieu Moy. Pinavm: a systemc front-end based on an executable intermediate representation. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 79–88. ACM, 2010.

[14] Weiwei Chen, Xu Han, and Rainer Dömer. May-happen-in-parallel analysis based on segment graphs for safe esl models. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 287. European Design and Automation Association, 2014.

[15] Anirudh Kaushik and Hiren D Patel. Systemc-clang: An open-source framework for analyzing mixed-abstraction systemc models. In *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, pages 1–8. IEEE, 2013.

[16] Quinlan and D.J. Compass user manual (2008). `http://www.rosecompiler.org/compass.pdf`. [Online; accessed 19-June-2019].

[17] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[18] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 15–24. ACM, 2017.

[21] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84. IEEE, 2016.

[22] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[23] Pascale Guerdoux-Jamet and Dominique Lavenier. Samba: hardware accelerator for biological sequence comparison. *Bioinformatics*, 13(6):609–615, 1997.

[24] Jason Chiang, Michael Studniberg, Jack Shaw, Stephen Seto, and Kevin Truong. Hardware accelerator for genomic sequence alignment. In *2006 International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 5787–5789. IEEE, 2016.

[25] Azzedine Boukerche, Jan M Correa, Alba Cristina Magalhaes Melo, and Ricardo P Jacobi. A hardware accelerator for the fast retrieval of dialign biological sequence alignments in linear space. *IEEE Transactions on Computers*, 59(6):808–821, 2010.

[26] Seamas McGettrick, Dermot Geraghty, and Ciaran McElroy. An fpga architecture for the pagerank eigenvector problem. In *2008 International Conference on Field Programmable Logic and Applications*, pages 523–526. IEEE, 2008.

[27] Osama G Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 228–235. IEEE, 2014.

[28] Svetlin A Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *2007 IEEE International Conference on Signal Processing and Communications*, pages 65–68. IEEE, 2007.

[29] Brahim Betkaoui, Yu Wang, David B Thomas, and Wayne Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, pages 8–15. IEEE, 2012.

[30] Yaman Umuroglu, Donn Morrison, and Magnus Jahre. Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2015.

[31] Qingbo Wang, Weirong Jiang, Yinglong Xia, and Viktor Prasanna. A message-passing multi-softcore architecture on fpga for breadth-first search. In *2010 International Conference on Field-Programmable Technology*, pages 70–77. IEEE, 2010.

[32] George Rosario Jagadeesh, Thambipillai Srikanthan, and CM Lim. Field programmable gate array-based acceleration of shortest-path computation. *IET computers & digital techniques*, 5(4):231–237, 2011.

[33] Michael DeLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E Uribe, F Thomas Jr, Andre DeHon, et al. Graphstep: A system architecture for sparse-graph algorithms. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 143–151. IEEE, 2006.

[34] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28. IEEE, 2014.

[35] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[36] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[37] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 30(2):305–324, 2018.

[38] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation: numerical methods*, volume 23. Prentice hall Englewood Cliffs, NJ, 1989.

[39] Joseph Gonzalez, Yucheng Low, and Carlos Guestrin. Residual splash for optimally parallelizing belief propagation. In *Artificial Intelligence and Statistics*, pages 177–184, 2009.

[40] Radford M Neal and Geoffrey E Hinton. A view of the em algorithm that justifies incremental, sparse, and other variants. In *Learning in graphical models*, pages 355–368. Springer, 1998.

[41] Arvind Arasu, Jasmine Novak, Andrew Tomkins, and John Tomlin. Pagerank computation and the structure of the web: Experiments and algorithms. In *Proceedings of the Eleventh International World Wide Web Conference, Poster Track*, pages 107–117, 2002.

[42] William G Macready, Athanassios G Siapas, and Stuart A Kauffman. Criticality and parallelism in combinatorial optimization. *Science*, 271(5245):56–59, 1996.

[43] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

[44] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, Steven Burns, and Ozcan Ozturk. Architectural requirements for energy efficient execution of graph analytics applications. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 676–681. IEEE, 2015.

[45] Hardware accelerator research program. `https://software.intel.com/en-us/hardware-accelerator-research-program`. [Online; accessed 19-June-2019].

[46] Stratus High-Level Synthesis. `https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html`. [Online; accessed 19-June-2019].

[47] Vivado Design Suite - HLx Editions. `https://www.xilinx.com/products/design-tools/vivado.html`. [Online; accessed 19-June-2019].

[48] Intel® Quartus® Prime Pro and Standard Edition Software User Guides. `https://www.intel.com/content/www/us/en/programmable/products/design-software/fpga-design/quartus-prime/user-guides.html`. [Online; accessed 19-June-2019].

[49] The LLVM Compiler Infrastructure. `https://llvm.org/`. [Online; accessed 19-June-2019].

[50] GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`. [Online; accessed 19-June-2019].

[51] Nikolai Sorokin. Implementation of high-speed fixed-point dividers on fpga. *Journal of Computer Science & Technology*, 6, 2006.

[52] Daniel Menard, Daniel Chillet, and Olivier Sentieys. Floating-to-fixed-point conversion for digital signal processors. *EURASIP journal on applied signal processing*, 2006:77–77, 2006.

[53] The industry's first floating-point fpga. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/po/bg-floating-point-fpga.pdf`. [Online; accessed 19-June-2019].

[54] Reduce power and cost by converting from floating point to fixed point. `https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf`. [Online; accessed 19-June-2019].

[55] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7):577–588, 2014.

[56] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[57] David Peleg. Distributed computing: A locality-sensitive approach. *SIAM journal on computing*, 01 2000.

[58] Yossi Shiloach and Uzi Vishkin. An o (log n) parallel connectivity algorithm. Technical report, Computer Science Department, Technion, 1980.

[59] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[60] Open SystemC Initiative. `http://www.systemc.org/`. [Online; accessed 19-June-2019].

[61] Adam Rose, Stuart Swan, John Pierce, Jean-Michel Fernandez, et al. Transaction level modeling in systemc. *Open SystemC Initiative*, 1(1.297), 2005.

[62] Philippe Coussy and Adam Morawiec. *High-level synthesis*, volume 1. Springer, 2010.

[63] Michael C McFarland, Alice C Parker, and Raul Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.

[64] Intel® Accelerator Functional Unit (AFU) Simulation Environment (ASE). `https://opae.github.io/0.13.1/docs/ase_userguide/ase_userguide.html`. [Online; accessed 19-June-2019].

[65] ISE Design Suite. `https://www.xilinx.com/products/design-tools/ise-design-suite.html`. [Online; accessed 19-June-2019].

[66] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.