

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

An Exploration of FPGAs as Accelerators for Graph Analysis via High-Level Synthesis

Pedro Filipe Silva



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Bispo

Second Supervisor: Nuno Paulino

October 29, 2021

An Exploration of FPGAs as Accelerators for Graph Analysis via High-Level Synthesis

Pedro Filipe Silva

Mestrado Integrado em Engenharia Informática e Computação

October 29, 2021

Abstract

Graph-based abstractions are common in several fields of research and industry. For instance, traffic navigation is one such field, in which a traffic network (often a road network) may be represented as a directed graph. Such graphs can be both quite large and complex to process; however, their analysis is very relevant for certain applications deployed, for instance, in High-Performance Computing (HPC) centres, making graph algorithms good candidates for hardware acceleration.

Our work focuses on exploring FPGA acceleration of graph algorithms, following the High-Level Synthesis (HLS) approach. Specifically, we target the Xilinx Alveo series of data-centre acceleration cards. Although our initial focus is on a traffic navigation system, the work applies to graph analysis in general. FPGAs are currently a trending topic in HPC due to several factors, such as increased energy efficiency, but still lag behind GPUs regarding adoption as accelerators. We coin the term *FPGA-unfriendliness* to refer to FPGA-related limitations such as limited memory bandwidth and low operating clock rates, which may significantly impact certain application domains – such as graph analysis.

We develop accelerators for three graph algorithms, two of them based on Xilinx Vitis Libraries kernels (Single-Source Shortest Paths – SSSP – and PageRank – PR), and one built from the ground up (Distinctiveness Centrality 1 – DC1). All accelerators are implemented via HLS. For DC1, we develop and present several variants, each corresponding to an optimisation of its predecessor. These variants are based on a novel formulation of DC1 (DC1-CSR) which is amenable to FPGA implementation, presenting properties such as compatibility with streaming and data-parallelism.

We compare against published results for SSSP and PR acceleration using traditional execution platforms (CPU and GPU), and against our own CPU C++ implementation for DC1. Our results indicate that FPGA acceleration of SSSP is not currently viable; that PR acceleration fares well against CPU execution and might be viable presently or in the near future; for DC1, we achieve significant improvement versus the C++ implementation (up to 25x speedup and 22x energy efficiency increase).

We conclude with a discussion on the current state of FPGA acceleration, and posit that HLS (and HLS-based high-level tooling) is key to advancing the field, especially in FPGA-unfriendly domains. Additionally, we propose a system for classification of HLS programming abstraction levels, which may provide some guidance regarding efforts on improving HLS programming tools and libraries.

Keywords: Field-Programmable Gate Arrays (FPGAs); Hardware Acceleration; High-Level Synthesis (HLS); Software Abstraction; Graph Analysis; Graph Algorithms; Graph Centrality; Single-Source Shortest Paths (SSSP); PageRank (PR); Distinctiveness Centrality (DC)

Resumo

As abstrações em grafos são comuns em vários campos de investigação e da indústria. A navegação em trânsito é um destes campos, no qual uma rede de trânsito (muitas vezes uma rede rodoviária) pode ser representada como um grafo dirigido. Tais grafos podem ser bastante extensos e possuir alta complexidade de processamento; no entanto, a sua análise é muito relevante para certas aplicações utilizadas, por exemplo, em centros de *High-Performance Computing* (HPC), tornando os algoritmos de grafos bons candidatos para aceleração via hardware.

O nosso trabalho foca-se na exploração da aceleração de algoritmos de grafos usando FPGAs, seguindo a abordagem *High-Level Synthesis* (HLS). Especificamente, o nosso alvo é a série Xilinx Alveo de placas de aceleração para centros de dados. Apesar do nosso foco inicial ser num sistema de navegação de trânsito, o trabalho é aplicável à análise de grafos no geral. Os FPGAs são atualmente um tópico em voga em HPC devido a vários fatores, como maior eficiência energética, mas ainda permanecem atrás dos GPUs no que toca à adoção como aceleradores. Criamos o termo *FPGA-unfriendliness* para nos referirmos a limitações relacionadas com FPGAs, tais como largura de banda de memória limitada e baixa frequência de relógio, que podem significativamente impactar certos domínios aplicacionais – como a análise de grafos.

Desenvolvemos aceleradores para três algoritmos de grafos, dois deles baseados em *kernels* das bibliotecas Xilinx Vitis (*Single-Source Shortest Paths* – SSSP – e *PageRank* – PR), sendo o último construído de raiz (*Distinctiveness Centrality 1* – DC1). Todos os aceleradores foram implementados via HLS. Para o DC1, desenvolvemos e apresentamos múltiplas variantes, cada uma correspondendo a uma otimização ao seu predecessor. Estas variantes são baseadas numa nova formulação do DC1 (DC1-CSR) que é adequada a implementação em FPGA, apresentando propriedades tais como compatibilidade com *streaming* e paralelismo ao nível de dados.

Comparamos contra resultados publicados para aceleração SSSP e PR usando plataformas de execução tradicionais (CPU e GPU), e contra a nossa própria implementação C++ para CPU no caso do DC1. Os nossos resultados indicam que a aceleração de SSSP usando FPGAs não é atualmente viável; que a aceleração de PR obtém resultados competitivos contra a execução em CPU e poderá ser viável atualmente ou no futuro próximo; para o DC1, atingimos melhorias significativas versus a implementação C++ (melhoria de performance até 25x e aumento da eficiência energética até 22x).

Concluimos com uma discussão sobre o estado atual da aceleração usando FPGAs, e postulamos que o HLS (e ferramentas de alto nível para desenvolvimento HLS) é central para avançar o campo, especialmente em domínios *FPGA-unfriendly*. Adicionalmente, propomos um sistema para classificação de níveis de abstração em programação HLS, o que poderá fornecer alguma orientação nos esforços para melhorar ferramentas e bibliotecas de programação HLS.

Palavras-Chave: Field-Programmable Gate Arrays (FPGAs); Aceleração Hardware; High-Level Synthesis (HLS); Abstrações Software; Análise de Grafos; Algoritmos de Grafos; Centralidade de Grafos; Single-Source Shortest Paths (SSSP); PageRank (PR); Distinctiveness Centrality (DC)

Agradecimentos

Às artes.

Aos meus professores.

Aos meus orientadores, pelo apoio insubstituível.

Aos amigos e colegas, por boas ideias e tempos agradáveis.

À minha família. Muito, muito obrigado.

“Um fogo-fátuo rútilo, talvez...”

Florbela Espanca

“You are not expected to understand this.”

John Lions, on an abstraction failure,
A Commentary on the Sixth Edition Unix Operating System

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Objectives	2
1.4	Contributions	3
1.5	Document Structure	4
2	Background	5
2.1	Origin and Fundamentals of FPGAs	5
2.2	High-Level Synthesis	7
2.3	FPGAs for Software Developers, and in High-Performance Computing	8
2.4	Centrality Measures	10
2.5	Single-Source Shortest Paths and Pagerank	11
2.5.1	SSSP	12
2.5.2	PageRank	12
2.6	Graph Representations, Programming Abstractions, and Frameworks	13
2.7	FPGA-Unfriendliness	15
3	Related Work	17
3.1	HLS-Based Work	18
3.2	Vendor Libraries	20
3.3	Analysis	21
4	Topics on High-Level Synthesis and FPGA Accelerator Boards	25
4.1	High-Level Synthesis	25
4.1.1	Dataflow	25
4.1.2	Unrolling	27
4.1.3	Pipelining	27
4.2	FPGA Accelerator Boards	29
4.2.1	Shells	29
4.2.2	Topology	29
4.2.3	HBM Considerations	31
5	Implementation	33
5.1	Kernel Runner	33
5.2	Single-Source Shortest Paths	34
5.3	PageRank	35
5.4	Distinctiveness Centrality 1	36

5.4.1	DC1 and its CSR formulation	36
5.4.2	DC1-CSR Algorithm	38
5.4.3	Baseline Kernel	40
5.4.4	Variants	42
5.4.5	Optimisations	42
6	Evaluation	51
6.1	Metrics	51
6.2	Experimental Setup	51
6.2.1	Alveo U250: Local Machine	51
6.2.2	Alveo U280: Remote Machine	52
6.2.3	Compilation	52
6.2.4	Datasets, Parameters, and Comparison Benchmarks	53
6.3	Results	54
6.3.1	SSSP versus High-Performance Frameworks	54
6.3.2	PageRank versus High-Performance Frameworks	56
6.3.3	Distinctiveness Centrality 1	63
7	Discussion and Conclusion	75
7.1	A Systematisation of HLS Abstractions and Optimisations	75
7.2	DC1-CSR Properties	76
7.2.1	Data-Parallelism	76
7.2.2	Streamability	77
7.2.3	Dynamicism	77
7.2.4	Possible Iterative Variants	78
7.3	Conclusion	78
7.4	Further Work	80
A	Distinctiveness Centrality 1 Kernel Variants	81
A.1	dc1	81
A.2	dc1_s	85
A.3	dc1_no_idx	85
A.4	dc1_deg_pow	85
A.5	dc1_no_col	87
A.6	dc1_unroll	87
B	Absolute Execution Metrics	93
B.1	SSSP	93
B.2	PR	93
B.3	DC1	94
	References	101

List of Figures

2.1	Architectural Diagram for a Typical Modern Platform FPGA Expansion Card . . .	6
2.2	A Xilinx PCIe FPGA Accelerator Card	9
2.3	An Intel PCIe FPGA Accelerator Card	10
2.4	A Sample from the Czechia Road Network Dataset	11
2.5	Betweenness Centrality Values for a Sample Graph	12
3.1	The ThunderGP Gather-Apply-Scatter Formulation	23
4.1	Dataflow: Example Dependencies	26
4.2	Dataflow: Example Serial Execution	26
4.3	Dataflow: Example Concurrent Execution	27
4.4	Pipelining: Example Serial Execution	28
4.5	Pipelining: Example Pipelined Execution	28
4.6	Topology for the Alveo U250 201830_2 Platform	30
4.7	Topology for the Alveo U250 gen3x16_base3 Platform	30
4.8	Topology for the Alveo U280 gen3x16_base3 Platform	30
5.1	Vectors V and C	37
5.2	Zero-Indexed Vectors V , C , and O , as well as Vector I (V Index) and S (C Index) .	38
5.3	Obtaining the Vector of Vertex Degrees, D , via C	38
6.1	Execution Time for SSSP – CPU (Zheng/AMD) over FPGA, per Dataset and Framework	55
6.2	Execution Time for SSSP – GPU (Graphit/Quadro) over FPGA, per Dataset and Framework	55
6.3	Execution Time for U250 PR – GPU (Graphit/Quadro) over FPGA, per Dataset and Framework	56
6.4	Execution Time for U250 PR – CPU (Zheng/Intel) over FPGA, per Dataset and Framework.	57
6.5	Execution Time for U250 PR – CPU (Zheng/AMD) over FPGA, per Dataset and Framework.	58
6.6	Execution Time for U280 PR, 2 Channels – CPU (Zheng/Intel) over FPGA, per Dataset and Framework	59
6.7	Execution Time for U280 PR, 2 Channels – CPU (Zheng/AMD) over FPGA, per Dataset and Framework	60
6.8	Execution time for U280 PR, 2 Channels – GPU (Graphit/Quadro) over FPGA, per Dataset and Framework.	60
6.9	Execution Time for U280 PR, 6 Channels – CPU (Zheng/AMD) over FPGA, per Dataset and Framework	61

6.10	Execution time for U280 PR, 6 Channels – GPU over FPGA, per Dataset and Framework.	61
6.11	Execution Time for U280 PR, Jumbo – GPU over FPGA, per Dataset and Framework	62
6.12	Execution Time for U280 PR, Jumbo – CPU (Zheng/Intel) over FPGA, per Dataset and Framework	62
6.13	Execution Time for U280 PR, Jumbo – CPU (Zheng/AMD) over FPGA, per Dataset and Framework	63
6.14	Kernel Time for DC1 – DDR4 over DDR1, per Dataset	64
6.15	Kernel Time for DC1 – DDR4ODP over DDR4, per Dataset	65
6.16	Kernel Time for DC1 – DDR4ODP over DDR1, per Dataset	65
6.17	Kernel Time for DC1 – DDR4ODPCB1 over DDR4ODP, per Dataset	66
6.18	Kernel Time for DC1 – DDR4ODPUCB1 over DDR4ODPCB1, per Dataset . . .	66
6.19	Kernel Time for DC1 – HBM1J7 over DDR1, per Dataset	67
6.20	Kernel Time for DC1 – HBM8 over HBM1J7, per Dataset	68
6.21	Kernel Time for DC1 – HBM8ODP over HBM8, per Dataset	68
6.22	Kernel Time for DC1 – HBM10ODP over HBM8ODP, per Dataset	69
6.23	Kernel Time for DC1 – HBM10ODPCB1 over HBM10ODP, per Dataset	69
6.24	Kernel Time for DC1 – HBM10ODPUCB1 over HBM10ODPCB1, per Dataset .	70
6.25	Kernel Time for DC1 – DDR4ODPUCB1 Excluding Degree Generation on the Enqueue Loop, over the Regular Variant, per Dataset	70
6.26	Kernel Time for DC1 – HBM10ODPUCB1 Excluding Degree Generation on the Enqueue Loop, over the Regular Variant, per Dataset	71
6.27	Total Execution Time and Total Energy Expenditure, Left to Right, for DC1 DDR1 – FPGA over CPU, per Dataset	72
6.28	Total Execution Time and Total Energy Expenditure, Left to Right, for DC1 DDR4ODP – FPGA over CPU, per Dataset	72
6.29	Total Execution Time and Total Energy Expenditure, Left to Right, for DC1 DDR4ODPUCB1 – FPGA over CPU, per Dataset	73
6.30	Total Execution Time for DC1 HBM10ODPUCB1 – FPGA over CPU, per Dataset	73
6.31	Cumulative Unique Page Requests For Writes vs. Number of Processed Edges for Datasets R4, RMAT, and US, for CPU DC1 Implementation	74
7.1	Possible Mapping of Abstraction Levels to Developer Knowledge Acquisition Effort Rate	77
A.1	Kernel Time and Total Execution Time vs. Number of Chunks per CU, for Variant DDRODPUCB and Dataset CO	86
A.2	Kernel Time and Total Execution Time vs. Number of Chunks per CU, for Variant DDRODPUCB and Dataset CO	87
A.3	Kernel Time and Total Execution Time Excluding Degree Queue Generation vs. Number of Chunks per CU, for Variant DDRODPUCB and Dataset CO	91
A.4	Kernel Time and Total Execution Time Excluding Degree Queue Generation vs. Number of Chunks per CU, for Variant DDRODPUCB and Dataset CO	92
B.1	Total Execution Time for all DDR1 Variants, in Progression Order	99
B.2	Total Execution Time for DDR1 D Optimisation Variants, in Progression Order .	99

List of Tables

2.1	LUT Mask for a 3-way XOR Operation	6
3.1	Other Frameworks (of non-HLS or Unknown Nature)	19
3.2	HLS-Based Graph Algorithm Work	20
3.3	Vitis HLS Libraries: Selected Graph Applications	21
5.1	PageRank Application Variants	36
5.2	Correspondence between DC1 Versions, Optimisations, and Kernel Versions . . .	49
6.1	Dataset Shorthands and Origin	53
6.2	Author-Algorithm-Dataset Correspondence	54
7.1	Classification System for HLS Programming Abstraction Levels	76
A.1	Resource usage for dc1, 1 CU, Alveo U250	84
A.2	Resource usage for dc1, 1 CU, Alveo U280	84
A.3	Resource usage for dc1_s, 4 CUs, Alveo U250	85
A.4	Resource usage for dc1_no_idx, 8 CUs, Alveo U280	85
A.5	Resource usage for dc1_deg_pow, 4 CUs, Alveo U250	85
A.6	Resource usage for dc1_deg_pow, 8 CUs, Alveo U280	86
A.7	Resource usage for dc1_no_col, 10 CUs, Alveo U280	87
A.8	Resource usage for dc1_unroll, 4 CUs, Alveo U250	90
A.9	Resource usage for dc1_unroll, 10 CUs, Alveo U280	91
B.1	Execution Metrics for SSSP	93
B.2	Execution Metrics for PR-DDR	93
B.3	Execution Metrics for PR-HBM2C	94
B.4	Execution Metrics for PR-HBM6C	94
B.5	Execution Metrics for PR-HBMJ	94
B.6	Execution Metrics for DDR1	94
B.7	Execution Metrics for DDR4	95
B.8	Execution Metrics for DDR4ODP	95
B.9	Execution Metrics for DDR4ODPCB1	95
B.10	Execution Metrics for DDR4ODPCUB1	96
B.11	Execution Metrics for DDR4ODPCUB1, Excluding Degree Calculation	96
B.12	Execution Metrics for HBM1J7	96
B.13	Execution Metrics for HBM8	97
B.14	Execution Metrics for HBM8ODP	97
B.15	Execution Metrics for HBM10ODP	97

B.16 Execution Metrics for HBM10ODPCB1	98
B.17 Execution Metrics for HBM10ODPUCB1	98
B.18 Execution Metrics for HBM10ODPUCB1, Excluding Degree Calculation	98

Abbreviations

ASIC	Application-Specific Integrated Circuit
BC	Betweenness Centrality
BRAM	Block Random-Access Memory
BSP	Bulk-Synchronous Parallel
CDFG	Control and Data Flow Graph
COO	Coordinate (sparse matrix format)
CSR	Compressed Sparse Row
DC	Distinctiveness Centrality
DRAM	Dynamic Random-Access Memory
DSL	Domain-Specific Language
DSP	Digital Signal Processing/Processor
EDA	Electronic Design Automation
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
GAS	Gather-Apply-Scatter
GPGPU	General-Purpose Graphics Processing Unit
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HLS	High-Level Synthesis
HMC	Hybrid Memory Cube
HPC	High-Performance Computing
IT4I	IT4Innovations
LB	Logic Block
LIL	Linked List (sparse matrix format)
LUT	Look-Up Table
PCIe	PCI-Express
PI	Programmable Interconnect
PR	PageRank
PROM	Programmable Read-Only Memory
RTL	Register-Transfer Level
SDRAM	Synchronous Dynamic Random-Access Memory
SLR	Super Logic Region
SSSP	Single-Source Shortest Paths
SRAM	Static Random-Access Memory
URAM	UltraRAM

Chapter 1

Introduction

1.1 Context

The graph – as defined in the field of discrete mathematics – is a common representational abstraction in all manner of application domains. Commonly, the abstract *object* a graph represents is a network, consisting of subjects and links between subjects [10]. Other objects possibly modelled by graphs include, for instance, the computational flow of a program, or data contained in tree structures (as a tree is an instance of a graph type).

Routing is the problem of assigning paths – or routes – to entities traversing a network, in such a way that the selection optimises one or more metrics. In the field of Computer Science, this often refers to the problem of data transferal within a packet-switched network, such as the Internet. Other important routing problems exist, however – such as those in the classes of *vehicle routing* problems (usually offline), or *route planning* problems (usually online); both are relevant to contemporary navigation research.

Traffic control is an active and prolific research field [32, 58]. IT4Innovations, a public supercomputing R&D institute linked to the Technical University of Ostrava, has developed and is developing a navigation system (referred to as *NavSys*) based on state-of-the-art research in traffic routing and supporting algorithms. The SPeCS laboratory of the Faculty of Engineering of the University of Porto, focused on analysis and optimisation of applications for special-purpose computing systems, has come to collaborate with IT4I in optimising this system.

The present work exists within the context of this collaboration. Both IT4I and SPeCS have become interested in exploring Field-Programmable Gate Array (FPGA) technology, especially High-Level Synthesis, both as a means of accelerating stages of the IT4I traffic navigation system, as well as for the possibility of graph analysis acceleration in general, as applicable to several other domains. We explore the routing issue in particular in our introduction, but expand our outlook significantly to *all* graph analysis as we analyse the background and state-of-the-art in FPGA graph processing in Chapter 2.

1.2 Motivation

Emerging navigation systems, linked to the *smart city* concept [65], must execute on large datasets, often under real-time constraints [19]. In addition, the problems and algorithms involved are often complex: *solving* route planning adequately, in one of its many formulations – for instance, to enable a smart city traffic planning system – is often computationally intensive [3] (and, thus, energy intensive). Issues in traffic routing and navigation are typically framed as graph analysis problems: the graph is often the perfect abstraction for a traffic network [70].

NavSys is a High-Performance Computing (HPC) traffic navigation application [65]. HPC projects place a heavy focus on efficiency, and “performance” often refers to execution time. However, in several fields, including HPC, energy efficiency is increasingly a primary metric, due to both environmental and engineering concerns [75, 80].

Field Programmable Gate Arrays, traditionally a niche technology [25] (as General-Purpose GPUs – GPGPUs – once were), have in recent years experienced a resurgence, aided in part by significant development of High-Level Synthesis (HLS) tools. HLS generates circuitry from general purpose programming languages, such as C, instead of Hardware Description Languages (HDLs) [50]. FPGA devices are often ranked between Application-Specific Integrated Circuits (ASICs) and CPUs in terms of both time- and energy-efficiency [114], and while the energy-efficiency gap has been becoming shorter in recent years – as FPGAs increase in complexity to support HPC/data-centre applications – they still hold the advantage in this regard [74]. As a trade-off for – and side-effect of – permitting reconfiguration, FPGAs are less computationally dense than both CPUs and GPUs [40].

However, it is common for graph analytic algorithms to present high memory to compute ratios as well as irregular memory access patterns, which pose issues for currently commonly-available FPGA platforms. As a result, the use of FPGAs to accelerate graph analytic workloads remains quite unexplored [5]; this includes research and published experience on the suitability of specific graph problems, or families of problems for FPGA acceleration. On a brighter note, a new generation of FPGA platforms equipped with High-Bandwidth Memory (HBM) is currently emerging. This technology provides much higher memory access bandwidths [15], making research towards FPGA graph processing all the more relevant.

The initial prompt for our exploration of FPGA/HLS graph analysis is the traffic routing problem, and our main contribution is directly applicable to the issue. However, we are also motivated by the problem of graph analysis in general, and a large portion of our work reflects this.

1.3 Objectives

The objective of this work is twofold. We wish to advance the state of the art by:

- Evaluating the viability of FPGAs as accelerators for graph analysis, especially centrality (we elaborate on centrality measures in Section 2.4);

- Determining the best approach to enable such an acceleration, taking into account the maturity of current frameworks, toolsets, and other supporting technology;
- Detailing the current and near-future state of HLS-based graph analysis on FPGAs, as well as HLS programming overall, as a guide for future researchers or developers.

Whereas, from an engineering perspective, we focus on graph analytic applications:

- Assembling a suite of graph analysis algorithms which provide good coverage of typical graph problems and applications, and attempting acceleration of said algorithms;
- Evaluating the (energy- and time-) performance of accelerated workloads on commonly used benchmark datasets;
- Evaluating the properties of any novel accelerated algorithms as a “triage test” of their eventual usefulness to NavSys and other applications.

1.4 Contributions

The main contributions we present in our work include:

- A novel evaluation of the performance of common graph algorithms (Single-Source Shortest Paths, PageRank), as accelerated using FPGA technology, versus traditional CPU and GPU accelerator frameworks;
- A novel formulation of a recent centrality metric (Distinctiveness Centrality [27]) suited specifically for FPGA execution;
- The first¹ accelerator for Distinctiveness Centrality which, by virtue of our formulation, possesses several useful properties such as data-parallelism, streamability, dynamicism, and, possibly, iterative refinement;
- A classification system for HLS abstraction levels, from a software engineering perspective, based on perceived optimisation difficulty;
- A crystallisation of the concept of *FPGA-unfriendliness*: the property carried by certain algorithms, programs, or entire application domains which makes their implementation/acceleration using FPGA technology difficult or poorly performant.

Our work has also resulted in the following additional contributions:

- A short paper entitled *Building Beyond HLS: Graph Analysis and Others*, presented at LATTE '21, an ASPLOS workshop [77]²;

¹To the best of our knowledge.

²Some content in Sections 2.3, 3.3, and 7.3 is derived from the author’s contribution to this work.

- A contribution to the PRACE project’s *Modern Accelerators* best-practice guide [6]³;
- An accepted submission to the 2021 International Conference on Field-Programmable Technology (FPT’21).

1.5 Document Structure

We have given a brief introduction to the context, motivation, objectives, and main contributions of our work in the current chapter. In Chapter 2, we present a background review of FPGA technology and graph analysis, both in general and in application to our work; Chapter 3 discusses recent work relevant to our objectives.

Chapter 4 gives information on specific FPGA and HLS topics which we believe is necessary before moving on to the description of our accelerated applications in Chapter 5.

The accelerated applications are: Single-Source Shortest Paths (SSSP), which gives the shortest (weighted) path between one source vertex and all other vertices in the graph; PageRank (PR), a widely used metric, mainly as a means of ranking webpage importance⁴ for search results, but used in other domains as well [29]; and Distinctiveness Centrality 1 (DC1), a novel centrality metric proposed by [27]. This set of three applications is representative of the varying kinds of algorithmic approaches found in graph analysis: SSSP is implemented as a path traversal algorithm, PR as an iterative convergence algorithm, and DC1 (or rather, our DC1-CSR formulation) as a streaming algorithm. Furthermore, IT4I has expressed interest in the acceleration of all three algorithms.

Chapter 6 details our evaluation of the generated accelerators. We conclude in Chapter 7 with our classification system for HLS programming abstraction levels, an outlook on FPGA graph acceleration based on our evaluation, an argument for further study of our Distinctiveness Centrality accelerator, and a brief synthesis on the current state of FPGAs in the software world as well as FPGA graph analysis (including an argument for focusing high-level tooling), as it relates to our work.

³Some content in Sections 2.1, 2.3, and 4.1 is derived from the author’s contribution to this work.

⁴PageRank is technically a centrality algorithm, but is not often referred to as such.

Chapter 2

Background

2.1 Origin and Fundamentals of FPGAs

Field-Programmable Gate Arrays are electronic devices which may be programmed. Unlike *software* programming, however, FPGA programming involves establishing connections within an array of logic blocks – in effect, allowing for programmable *hardware* able to implement any circuit, within resource constraints.

FPGA technology traces its origin to the first fuse-based (and, as a result, one-time) field-programmable devices created in the 1960s [46]. Hardware programmability evolved into mask- or fuse-based PROM solutions, to PLA and PAL devices with custom programmable AND (and in the former, OR) logic planes, to the first commercial FPGA system (the XC2064, created by Xilinx in collaboration with Seiko [81, 1]), arriving at the current Static Random-Access Memory (SRAM)-based FPGA [46].

Two factors led to the initial success of FPGA technology: lower overall costs for low-volume production, and rapid design/prototyping capabilities. FPGAs thus became a mainstay of the hardware development community, accompanying in tandem (and benefiting from) development advances in Electronic Design Automation (EDA), such as logic synthesis, which by the end of the 1990s had become required for all but the simplest projects. As FPGA usage grew to encompass increasingly compute-intensive applications (mostly due to the Internet-driven growth of the communications industry), vendors began to incorporate discrete logic blocks, such as large on-chip memories and microprocessors – thus shifting away from the pure “gate array” and giving rise to the modern *platform* FPGA [81].

Figure 2.1 shows the design architecture for a modern platform FPGA processor contained in a PC expansion card. What follows pertains to this particular type of FPGA: both historical and contemporary implementations are far from uniform [46]. Note the prevalence of *reconfigurable logic blocks* (LBs): these are the programmable functional building blocks of an FPGA. Each logic block consists of a programmable operator, implemented via one or more Lookup Tables (LUTs) and flip-flops acting as registers. LUTs amount to nothing more than small bit-addressable memory cells: they implement logic functions by simply storing their results in the memory address

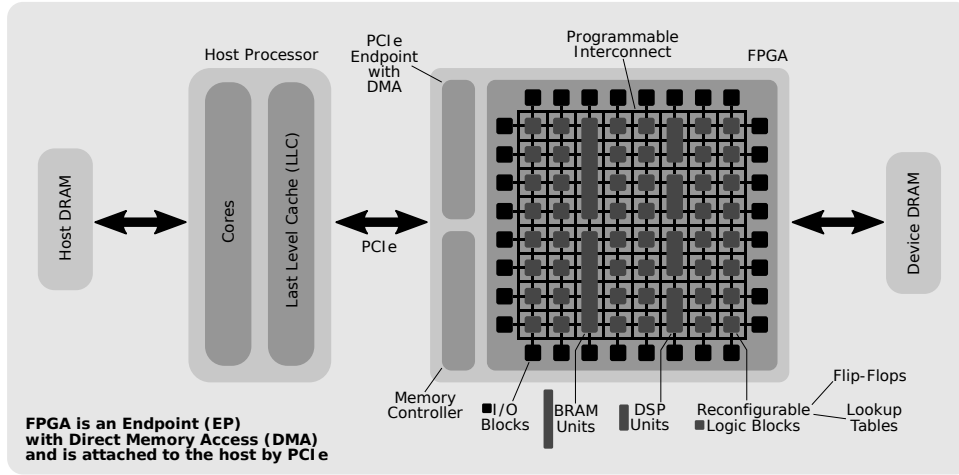


Figure 2.1: Architectural Diagram for a Typical Modern Platform FPGA Expansion Card

(Source: [5])

corresponding to the values of its operands [46].

LUTs may implement logic functions with an arbitrary number of operands, dependant on address space. Table 2.1 shows an example configuration for a LUT (termed a LUT mask) implementing a 3-way XOR operation; this equates, by definition, to the operation's truth table.

I/O blocks, which are placed at the periphery of the chip, allow the FPGA to communicate with the outside world. These are also programmable to serve as inputs *or* outputs, and may be directed to follow complex protocols [87].

The Programmable Interconnect (PI) allows the FPGA to communicate *internally*. It is reconfigurable, allowing a custom network to be built between functional elements [46].

LBs, I/O blocks, and the PI are the basic building blocks of the FPGA, and their conjunction is often named the FPGA *fabric* [87].

Also depicted are two types of non-cache memory. Block RAM (BRAM) is on-chip, high-throughput data storage (local memory), usually numbering in the tens of megabytes. Dynamic RAM (DRAM)¹ is larger, slower, off-chip data storage (global memory). One of the main challenges to FPGA implementation of many algorithms, including *graph* algorithms, is insufficient bandwidth between DRAM and the FPGA chip: random memory accesses are costly, and the

¹A technology also often used in modern PCs, in the form of *Synchronous* Dynamic RAM (SDRAM).

Table 2.1: LUT Mask for a 3-way XOR Operation

Addr. bit 1	Addr. bit 2	Addr. bit 3	Result
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

limited on-chip memory may often become depleted while waiting on data transfers [5]. New technologies, such as HBM, may alleviate these issues, as they offer much greater bandwidths, at storage capacities close to those of DRAM solutions. The card depicted in Figure 2.1, for instance (Alveo U280), includes a total of 8 GB *on-chip* HBM2, with a maximum bandwidth of 460 GB/s. [91].

Note additionally the presence of on-chip Digital Signal Processing (DSP) units, which are a common source of optimisation in data-centre platform FPGAs.

2.2 High-Level Synthesis

The shift in strategy that led to the platform FPGA was a precursor to more recent developments in the FPGA field. The most disruptive of these is likely the rise to maturity of High-Level Synthesis.

HLS was, for many years, referred to under a different name – *behavioural synthesis*, in contrast with the (mostly) RTL-based *functional* synthesis approach (or logic synthesis), common today via Verilog and VHDL compilers [31]. HLS is thus, in essence, the creation of a functional specification from a behavioural one, expressed in a general purpose programming language. To simplify, HLS allows hardware to be created from software.

Research in HLS, although prompted by earlier notions of compiling an instruction set specification into hardware [31], began in earnest in the 1980s [53]. Adoption of these early tools was limited by the parallel rise to prominence of logic synthesis, among other factors. A later phase, mostly driven by major players in the EDA industry, failed worse: now that logic synthesis had become well established, HLS tools of the time could not provide a better value proposition to users of the former, and vendors neglected to market to other potential customers (such as software developers) [53]. This failure led to a significant shift in the industry, leading even to the renaming of the technology and thus establishing current “High-Level Synthesis” [56].

Today, HLS tools are rising to the spotlight. High-level synthesised hardware is often quite competitive with RTL designs, and, even when not, allows hardware engineers to work more productively [47]. Software engineers, on the other hand, are adopting this generation of HLS, especially for FPGA-based acceleration. Higher-level tools are emerging, such as PYNQ [92], which exposes FPGA *overlays*² as Python libraries [43].

HLS tools compile to an RTL description of the target circuit. It is then the responsibility of logic synthesis tools to generate the actual hardware. In [16], the authors separate the HLS flow into a series of steps, starting at (user) specification and ending at logic synthesis:

1. *Compilation*: The transformation of the user functional specification (*e.g.*, in C) into a formal model. This often takes the form of a Control and Data Flow Graph (CDFG), where data dependencies are made explicit via *basic blocks* (straight-line code sequences), while the program control flow is coded in the directed edges between these blocks. This model

²A domain-specific term for program bitstreams, *i.e* the binary that is applied to an FPGA to construct a specific circuit, especially when provided as a higher-level abstraction such as an API.

allows for the representation of any input program and permits automatic or manual (user-driven) structural optimisations.

It is also during this stage that typically software-type optimisations come into play, such as dead-code elimination and constant folding. Along with these, many other hardware-specific optimisations are also possible [59].

2. *Allocation*: The selection of hardware resources (such as LUTs, interconnects, or, in platform FPGAs, devices like DSPs) deemed necessary for synthesis of the program.
3. *Scheduling*: Mapping of operations to one, or several, device clock cycles. Parallelism is exposed in this stage.
4. *Binding*: Mapping of variables to storage units, operations to generated or physical functional units, and transfers to transfer units, while safeguarding (and taking advantage of) dependency, lifetime, and communication requirements.
5. *Generation*: The creation of an RTL description of the hardware using the data provided by the prior three stages.

While the landscape of HLS research is lively [59], the commercial FPGA market is dominated by Xilinx (which was recently acquired by AMD [2]) and Intel [79] – meaning, due to the tendency for vendor control of FPGA EDA tools [81], that Xilinx and Intel software tools are the predominant FPGA HLS development platforms.

Both vendor toolsets compile C-like languages to RTL [101, 38], but alternative approaches exist. For instance, Maxeler’s MaxJ is a dataflow-oriented Java-based language (which compiles to VHDL) [44], while Calyx is an intermediate representation for FPGAs meant as a substitute compilation target [60].

2.3 FPGAs for Software Developers, and in High-Performance Computing

Today’s FPGA applications can broadly be grouped in two categories: traditional embedded devices (now being updated as concepts such as edge computing emerge), and HPC acceleration devices. This paradigm treats FPGA platforms as hardware accelerators – just like GPGPUs or the newer tensor processing devices – which are controlled by a host device (most often a PC). These platforms thus take the form of an expansion card, connected *e.g.*, via PCIe. Figures 2.2 and 2.3 show instances of such cards including FPGAs from the major vendors (Xilinx and Intel, respectively).

FPGA vendors have only recently begun exploring the HPC market [48]. Their success thus far was likely enabled by the aforementioned shift in HLS development and marketing; indeed, the existence of HLS tools usable by software engineers and domain experts was certainly the catalyst [48, 14] allowing FPGAs to take root in the community. GPGPUs have hardly been dethroned,



Figure 2.2: A Xilinx PCIe FPGA Accelerator Card

(Source: [88])

however: FPGAs are clearly superior in some applications but clearly inferior in others, allowing inertia to keep the *status quo* – although the panorama may be changing, driven by innovation in memory technologies [48, 15].

Factors of interest in FPGA technology include:

1. Increased energy efficiency [9, 57, 85, 48].
2. Perceived present or future time efficiency increase *vs.* traditional processing systems [57].
3. Interest in or perceived architectural advantages *vs.* traditional processing systems [9, 48].
4. Interest in reconfigurable heterogeneous computing systems [57, 85].

As mentioned in Section 2.2, the market is currently dominated by Xilinx and Intel. The flagship software-centric development platform for Xilinx is named Vitis, while Intel offers Quartus Prime. Both platforms prescribe OpenCL for host-side code; however, while Quartus Prime makes use of OpenCL for device-side/kernel code, Xilinx recommends its own HLS framework for Vitis (while still supporting OpenCL) [9].

The current panorama points to increasing adoption of FPGA accelerator technology³. The possible *extent* of this adoption, however, is yet to be known. While recent literature and studies cast a favourable look at HLS tools and the role they have played – and likely shall continue to play – in bringing FPGA acceleration closer to software and HPC developers [47], the approach is not considered perfect. Some referenced issues include:

1. Difficulty in implementation *vs.* GPU systems [57].

³The actions of major market players seem to support this: we refer back to the acquisition of Xilinx by AMD, as well as the earlier acquisition of Altera by Intel.



Figure 2.3: An Intel PCIe FPGA Accelerator Card

(Source: [36])

2. Lack of portability between FPGA vendors [85].
3. Optimisation challenges [85].
4. Reduced quality-of-results, in some instances, vs. RTL tools [47].
5. Failure of abstraction in some instances (as we ourselves shall argue).

2.4 Centrality Measures

We have previously mentioned that traffic navigation algorithms tend to make use of graph abstractions. The present work does not constitute an exception – NavSys works extensively with graphs derived from real-world road network data. Figure 2.4 shows a sample from a Czechia road network dataset ceded by IT4I. As can be seen, nodes represent junction, endpoints, or other features, while edges represent roads.

A key component of the framework is Betweenness Centrality (BC). This is a metric of graph node centrality (importance) which requires *global* information: the BC value for each node depends on the shortest paths between other nodes. This makes BC, a very popular metric for contemporary graph analysis, hard to compute both in terms of time and memory requirements, and there is abundant research on approaches attempting to minimise this issue [54]. Figure 2.5 shows BC values for a sample graph (where blue is higher).

The NavSys segment in [65] provides a high-level description of the NavSys flow, including the use of BC. NavSys uses BC to detect important nodes in a road network as a means of informing on graph edge weights, choosing viable paths for navigation, and selecting suitable targets for pre-computation.



Figure 2.4: A Sample from the Czechia Road Network Dataset

(Dataset Source: [82])

Several variants of BC exist. For one such variant, given by [8], the BC value for a node v is given by:

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where V is the set of nodes in the graph, σ_{st} is the number of shortest paths between nodes s and t , and $\sigma_{st}(v)$ is the number of said paths passing through v , with the caveat that if $s = t$, then $\sigma_{st} = 1$, and if $v \in s, t$, then $\sigma_{st}(v) = 0$.

Recently, a new family of metrics has been introduced: Distinctiveness Centrality (DC). DC is currently being studied for inclusion in NavSys, or other projects at IT4I. There exist several different metrics for DC, although all are based on the notion of prioritising nodes which function as gateways to lower-connected regions [27].

DC is a possible candidate for replacement of BC in the NavSys flow, provided its output is proven adequate and its performance better than that obtained by the current method. More details on one metric, Distinctiveness Centrality 1, are given in Section 5.4.

2.5 Single-Source Shortest Paths and Pagerank

Several other graph problems and algorithms exist, of course (we list some in Chapter 3). Of note for this work are Single-Source Shortest Paths (SSSP) and PageRank (PR).

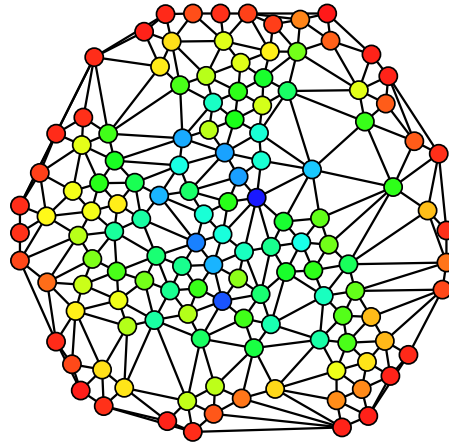


Figure 2.5: Betweenness Centrality Values for a Sample Graph

(Source: [13])

2.5.1 SSSP

SSSP refers to the problem of finding the shortest (weighted) paths from a single source vertex to all other vertices in the graph.

Several algorithms exist to solve this problem. Among the most well-known are Dijkstra's algorithm and its heuristic extension A*, which requires non-negative edge weights; and the Bellman-Ford algorithm, which supports any real-valued edge weights [22].

SSSP is clearly a fundamental problem, and as such it is relevant for many different applications, including routing [4].

2.5.2 PageRank

PageRank was devised as a metric of webpage importance to improve search result relevance. Many differing formulations of PR exist, but the most common is based on the random (web) surfer model [29, 26]. In this formulation, the PR value of a vertex (webpage) is given by the frequency of visits to that webpage during an infinite random walk of the World Wide Web via hyperlinks [26]. As such, vertices are more important if many other vertices point to them, provided these other vertices are also themselves important. This gives us an ideal recursive formulation of PR as

$$R_j = \sum_i R_i h_{ij}, \quad (2.1)$$

where i and j are webpages in the Web (or vertices in the graph view of the Web), and h_{ij} is the probability of a random surfer moving from i to j [26].

In reality, the model is slightly different, to prevent the surfer from getting trapped in a page or set of pages with no outgoing hyperlinks. A *damping factor*, α , is introduced. The surfer clicks a hyperlink on the current page with probability α , or manually moves to another random page on

the Web with probability $1 - \alpha$ [26]. This damping factor is commonly set to 0.85 (which may result in strong correlation to vertex degree [29]).

2.6 Graph Representations, Programming Abstractions, and Frameworks

Graphs are diverse. They may vary, for instance, in size, sparsity, and overall connectivity. On the other hand, graph accelerators themselves are also quite diverse. As a result, there are many different data structures used to represent graphs.

We may divide graph representation types into *logical* and *physical* categories. Physical graph representations may entail some form of compression of the logical data structure, for instance. Note that a graph may carry many different types of data, in many different forms, which may be transmitted in- or out-of-band – *e.g.*, by being directed, or undirected, and weighted, or unweighted. Often, by default, a logical representation may be safely assumed to be directed.

Logical graph representations include [68]:

- *Adjacency matrix*: A $V \times V$ matrix where each (i, j) entry is non-zero if there exists an edge between nodes i and j , and zero otherwise. Note that this format is especially amenable to storing edge weights.
- *Edge list*: A set of (i, j) tuples representing the existence of an edge between vertices i and j .
- *Neighbour (or adjacency) list*: $|V|$ sets, where set i gives the neighbours of node i .
- *Procedural*: A set of instructions which, when executed, construct the graph. For instance, a program which calls into a graph library permitting programmatic construction of graphs.

Road network graphs tend to be sparse: for instance, a junction will likely have a number of connections in the single-digit range. For some logical formats, their physical representation may follow directly; sparse graphs, however, when represented in a denser format such as adjacency matrix, often benefit from a more elaborate physical representation *e.g.*, for compression, as mentioned above.

Literature on FPGA graph acceleration tends to use some compressed form of the adjacency matrix. Physical formats thus include [72]:

- *Coordinate (COO)*: A set of three lists. For each non-zero entry at coordinates (i, j) with value k , i , j , and k are pushed onto each list. Significantly compresses the matrix while preserving readability.
- *Compressed Sparse Row (CSR)*: Also a set of three lists. The *val* and *col* lists contain the values and column indexes of non-zero entries, respectively. The *indexes* list contains, for each row, the index of its first element, with an additional element to terminate the last row

(which should be equal to the number of non-zero entries). In essence, to obtain values from a row i , one looks at the values (s, e) in the i -th and $(i + 1)$ -th entries in the *indexes* list, respectively, and extracts the range $[s, e[$ from *val*. Meant for optimising computational performance. See Subsection 5.4.1 for a constructive definition of CSR.

- *Linked List Matrix (LIL)*: Two lists, of size equal to the number of rows/vertices, containing the column and value of a non-zero entry respectively, for each row. Specifically meant for simple incremental construction of matrices via indexing and slicing operations; not quite storage- or computationally-optimised.

Efficient execution of algorithms often entails a distributed or parallel approach, requiring specialised knowledge and longer development cycles. In graph analysis, the problem is exacerbated further by the sheer diversity of computational patterns in play. To ameliorate this issue, several *programming abstractions* (which we also refer to as *paradigms*, or *models*) have been devised. When using such an abstraction, it is the job of the programmer to express the desired graph algorithm in the given model. Implementations of the model (which we refer to as *frameworks*⁴) then map the computation to any supported execution environment [41]. In turn, certain execution environments may be more amenable to certain programming models.

We give a broad overview of paradigms mentioned in Chapter 3:

- *Vertex-Centric*: Expresses computations from the point of view of a vertex. Each vertex has access to its ID, value, edges, and edge weights, and may send messages to its neighbours, as well as alter its own state as the result of a message. All vertices update in parallel during one iteration, termed *superstep*. This paradigm is quite popular, and (or perhaps due to being) very unrestrictive in the problems it allows to model. [41].
- *Gather-Apply-Scatter (GAS)*: A decomposition of the vertex-centric model into two phases, originally meant to better handle power-law graphs⁵. The *gather* stage is applied in parallel over all edges, grouped by destination vertex. The *sum* stage reduces the results of the gather stage for each destination vertex, and is sometimes grouped with the former. The *apply* stage uses this information to alter the vertex state, while in the *scatter* stage (which may be omitted), this state is propagated to neighbours. The value reduction in the sum stage restricts the kinds of computations that may be naturally expressed in the GAS model; there exists, however, a low-performant workaround to emulate the vertex-centric model. Iterations are also termed supersteps [41].
- *Bulk-Synchronous Parallel (BSP)*: Similarly to GAS, a three-phase iterative model. The phases run in parallel: the first phase comprises local (*e.g.*, vertex/edge-centric) computation; the second phase allows processes to communicate; the third phase is a synchronisation barrier [5].

⁴Such frameworks often handle graph representation (and partitioning) themselves.

⁵Graphs in which the majority of nodes have few neighbours, while a small number of nodes has a very high number of neighbours [41].

- *Edge-Centric (Streaming)*: Edges are streamed into a computational kernel, in order, and processed (conceptually) sequentially. For some applications, this may be a natural model for FPGA implementation; in others, however, the sequential access pattern may significantly increase computational demand [5].

These different graph representations and paradigms are leveraged by *graph processing frameworks*: tools for graph algorithm execution which leverage different execution platforms for enhanced performance and present a usable API to the domain-expert [71].

Chapter 3 explores FPGA-oriented frameworks. As for the more common platforms (CPU and GPU), we give examples (which are later compared against in Section 6.3):

- Ligra, a CPU-based framework focused on graph traversal algorithms [76];
- Graphit, a toolset based on compiling a domain-specific language [108];
- Galois, focusing on providing a C++ abstraction layer for parallel programming patterns, especially in algorithms which operate on complex structures – such as graphs [64];
- Gunrock, a GPU graph library following the BSP model [84];
- SEP-Graph, a GPU vertex-centric modeswitching framework [83];
- GSWITCH, a GPU-based BSP abstraction layer with embedded autotuning [55].

2.7 FPGA-Unfriendliness

We have coined the term “FPGA-Unfriendliness“ in [77]. This is simply a crystallisation of a well-known concept: that implementation of certain algorithms, programs, (or entire application domains) using FPGA technology is difficult and/or poorly performant versus traditional execution platforms.

Graph analysis is by-and-large an unfriendly domain⁶. Low memory bandwidths, low amounts of fast local memory, and low clock rates do not mesh well with graph applications, which use complex data structures and are often dependant on random memory access.

Several factors related to FPGA technology and its ecosystem contribute to unfriendliness:

1. Hardware limitations (*e.g.*, low clock rates, limited global memory bandwidth, limited local memory capacity);
2. Synthesis limitations: as we focus on HLS, this mostly includes failings in mapping and abstraction between the high-level language source (*e.g.*, C/C++) and the low-level target (*e.g.*, Verilog), but could also refer to low abstraction levels hindering the implementation of complex algorithms when working at the RTL level;

⁶See [5, 77] for further details.

3. Software/IP limitations: lack of dedicated frameworks/libraries for unfriendly algorithms; libraries not sufficiently optimised; missing features in runtime, shell/board support packages, or development environments.

Moving forward, the inherent unfriendliness of graph analysis, and the quest for methods to reduce it, will be an implied factor in our work.

Chapter 3

Related Work

We now compile and analyse recent related work (from 2015 onward) on FPGA graph acceleration. Note that we focus on graph *analytic* algorithms, therefore excluding other related graph accelerators such as format converters and compressors. On the other hand, we include accelerators for applications which map directly to graph analytic algorithms, such as sparse matrix multiplication.

We additionally distinguish non-HLS (RTL) implementations (or literature unclear on implementation) from literature describing an HLS solution or framework. While many conclusions taken from these systems may also be valid for HLS work, focusing on the latter is preferable, as this work does not descend into the RTL level.

Frameworks based on the user programming at low-level (*e.g.*, RTL, or using a soft processor’s¹ custom instruction set) are excluded, as well as frameworks unclear on the method of programming. Additionally, many works focus on multiple issues, while not being presented as “frameworks” – instead opting to study aspects such as work-scheduling or partitioning more deeply. In many instances, these works do not properly describe usage, and are thus excluded from our analysis; we have included some on a case-by-case basis, when we considered they had a minimal degree of focus on usability.

We exclude non-HLS work from our detailed analysis below, but nonetheless it is compiled in Table 3.1, so as to allow us to draw high-level conclusions.

Section 3.1 and Table 3.2 compile information (including a detailed analysis) on HLS-based work. Section 3.2 and Table 3.3 compile information on vendor-provided HLS libraries.

Regarding all three sections:

- “N/A” means “not applicable”; “N/D” means “none declared”.
- “Framework usage” shows a brief example on how a framework may be used (if the entry constitutes a framework).
- Column “Problems” shows the problems explicitly addressed in the work; these include:

¹A processor fully realisable using logic synthesis. Commonly, an *FPGA* implementation of such a processor.

- *BFS*: Breadth-First Search.
- *SSSP*: Single-Source Shortest Path.
- *PR*: Pagerank.
- *(W)CC*: (Weighted) Connected Components.
- *CCe*: Closeness Centrality.
- *SpMV*: Sparse Matrix-Vector Multiplication.
- *TRW-S*: Tree Reweighted Message Passing.
- *CNN*: Convolutional Neural Network.
- *TC*: Triangle Counting.
- *SL*: Similarity.

3.1 HLS-Based Work

GraphSoC A custom soft-processor targeting graph algorithms framed in the vertex-centric model. The user specifies custom implementations for processor instructions `send`, `receive`, `update`, and `accum` via C++ API calls which are then put through a purpose-built compiler to generate processor microcode. The generated processor is then able to execute the graph algorithm as defined by the four instructions. An approach similar to others, such as [62] (see Table 3.1), but unusual both in using HLS to define the processor and in permitting the user to implement the instructions using a general-purpose language.

Spector A set of OpenCL FPGA-focused benchmarks. Provides extensive parametrisation. Declared objective is to permit research on design of FPGA algorithms and design space exploration. To this end, the algorithms are not overly tuned to FPGA execution, although there are architecture-specific optimisations. Both graph algorithms mentioned in Table 3.2 are based on the architecture-agnostic OpenDwarfs suite, which also includes rudimentary FPGA-specific optimisations [45]. Each implementation comes equipped with a set of knobs allowing the aforementioned parametrisation: for instance, the BFS algorithm has an *unroll factor* knob for permitting a loop to process multiple edges in parallel.

OBFS OpenCL single-problem tool for BFS. Focus is on exploiting latent parallelism in the algorithm, via pipelining and optimising costly memory accesses via graph reordering; and improving performance via memory access batching and coalescence. Implemented using OpenCL as a proof-of-concept showcasing the design productivity and maintainability benefits of HLS vs. RTL programming.

²These hardware blocks, are, in fact, implemented using MaxJ, a HLS approach. However, MaxJ strays from the general-purpose language format (by focusing on stream computations); we deem both this and the aforementioned hardware block aspect sufficient cause to not consider this work in our main analysis.

Table 3.1: Other Frameworks (of non-HLS or Unknown Nature)

Author-year	System name	Framework usage	Problems	Paradigm	$ V $	$ E $
Nurvitadhi et. al. (2014) [62]	GraphGen	Soft-processor (user defines custom instructions via RTL)	TRW-S, CNN	Vertex-centric	110.6k	221.2k
Dai et. al. (2016) [17]	FPGP	<i>User defined processing kernel</i>	BFS	N/D	1.4B	6.6B
Engelhardt and So (2016) [24]	GraVF	User-defined apply and scatter RTL kernels	PR, BFS, SSSP, CC	Vertex-centric	128k	512k
Oguntebi and Olukotun (2016) [63]	GraphOps	Composable, parametrisable hardware blocks ²	PR, SpMV, SSSP, others	Stream-centric	16M	128M
Zhou et. al. (2016) [110]	N/D	<i>User-defined scatter and gather kernels</i>	SSSP, WCC, MST	Edge-centric	4.7M	65.8M
Dai et. al. (2017) [18]	ForeGraph	<i>User defined processing elements</i>	PR, BFS, WCC	N/D	1.4B	6.6B
Zhou and Prasanna (2017) [113]	N/D	<i>User-defined scatter and gather kernels</i>	BFS, SSSP	Mode-switching edge-centric and vertex-centric	10M	160M
Zhou et. al. (2018) [112]	N/D	<i>User-specified edge-centric algorithms (and resource constraints)</i>	PR, SpMV	Edge-centric	41.6M	1.5B
Engelhardt and So (2019) [23]	GraVF-M	User-defined gather, apply, and scatter RTL kernels	PR, BFS, WCC	Gather-Apply-Scatter	N/D	N/D
Shao et. al. (2019) [73]	FabGraph	<i>User-defined processing pipeline</i>	PR, BFS	N/D	4.9M	69.0M
Zhou et. al. (2019) [111]	HitGraph	User-specified RTL edge-centric algorithms (and resource constraints)	PR, SSSP, WCC, SpMV	Edge-centric	41.6M	1.5B
Yang et. al. (2020) [106]	N/D	<i>User-defined vertex- and edge-oriented processing pipelines</i>	PR, BFS, WCC	Mode-switching edge-centric and vertex-centric	4.9M	69.0M

Gondhalekar and Feng (2020) OpenCL single-problem tool for BFS. Focus is on sparse graph-specific optimisations, and mitigating the performance impact from irregular memory access patterns typical in sparse graph algorithms. A queue-based, BRAM vertex-centric implementation meant to execute as a single kernel to avoid launch overhead. Key concepts are queue implementation as a bitmap and a focus on the necessity to restructure computations to better suit FPGA execution.

Hosseiniabady and Nunez-Yanez (2020) A Xilinx HLS implementation of SpMV. Focuses on modifying the common Compressed Sparse Row (CSR) format in order to permit a streaming

Table 3.2: HLS-Based Graph Algorithm Work

Author-year	System name	Framework usage	Problems	Paradigm	$ V $	$ E $
Kapre (2015) [42]	GraphSoC	Soft-processor (user defines custom instructions using C++)	SpMV	Vertex- centric	17.8k	126.2k
Gautier et. al. (2016) [28]	Spector	N/A	BFS, SpMV, others	N/D	N/A	N/A
Liu et. al. (2019) [51]	OBFS	N/A	BFS	N/D	4.9M	69.0M
Gondhalekar and Feng (2020) [30]	N/D	N/A	BFS	Vertex- centric	2.0M	3.0M
Hosseinabady and Nunez-Yanez (2020) [33]	N/D	N/A	SpMV	Edge- centric	39.9k	197.3k (2.9M)
Chen et. al. (2021 preprint) [12]	ThunderGP	User-defined gather, apply, and scatter kernels	CCe, PR, BFS, SpMV, others	Gather- apply- scatter	21.3M	265.0M (520.5M)

version of the SpMV kernel loop to be flattened, pipelined, and unrolled. Includes a discussion on load-balancing via matrix partitioning.

ThunderGP A Gather-Apply-Scatter (GAS) framework. Follows an earlier work by many of the same authors under an edge-centric approach [11]. The main focus is on providing a platform which requires a low degree of hardware expertise to use, and reducing development time. The onus is on the user to implement custom gather, apply, and scatter kernels, which are then compiled into the framework library to generate a hardware accelerator. Heavy emphasis on heuristics and Design Space Exploration to produce an optimised design relative to the algorithm and target platform. Graph partitioning is automatic. Relies on using global memory to store vertex data, allowing some pipelining optimisations, but uses on-chip memory for buffering and caching.

3.2 Vendor Libraries

While Intel provides several RTL-level IP core libraries [34], a FPGA-specific HLS math library [37] and several source-code libraries for CPUs and GPUs under its oneAPI model (which supports CPUs, GPUs, and FPGAs [35]), support for source-level domain-specific HLS libraries appears lacking. The focus for Intel at the moment appears to be integration at the IP level with possible configuration via APIs, although this may change in the future [39].

In contrast, Xilinx offers their own HLS accelerated math library [101], *as well as* a suite of HLS-level domain-specific libraries under the Vitis Libraries [103] umbrella. Table 3.3 shows a selection of graph-related work available mostly under the “Vitis Graph Library” moniker, as of version 2021.1. The SSSP and PR applications are further described in Chapter 5, as we make use of them.

Table 3.3: Vitis HLS Libraries: Selected Graph Applications

Library	Usage	Problem
Graph	Level 2, where the provided kernel is used as-is or modified; or Level 3, where the kernel is abstracted away via a host-side API	BFS, SSSP, CC, TC, PR, SL, others
SPARSE	Level 2	SpMV

3.3 Analysis

It comes into view immediately that the number of non-HLS-based frameworks is noticeably greater than that of their HLS counterparts. This may indicate one or two things: (1) that HLS is not yet viewed as fit for the task; or (2) that RTL development still carries a large momentum in the FPGA world, when compared to HLS. Regardless, recent literature argues that HLS is indeed ready, with some caveats [47], and recent HLS frameworks such as ThunderGP claim better results when compared to RTL frameworks [12] (this does not preclude potential quality-of-result issues vs. RTL, as mentioned in Section 2.3).

Some works present multi-FPGA implementations. We deem this acceptable in our analysis – our work will likely focus on single-FPGA implementations, but knowledge of the general structure of multi-FPGA works remains relevant.

We have presented a heavily constrained analysis, in order to fit our objectives³. However, in general, most works on graph acceleration in FPGAs possess at least one of the following attributes:

1. Tackling a single problem.
2. Not having available, runnable code.
3. Not being HLS-based, or being unclear about implementation.

When analysing non-HLS literature in particular (see Table 3.1), a key take-away is that the majority of works *does* focus on a single problem, and this tends to be a variation of the shortest-path problem. This perhaps indicates that pure RTL designs are hitting an abstraction limit, where other kinds of computations are very difficult to model.

A major aspect of this work, and the cause of our focus on HLS, is to validate the readiness of FPGA technology as graph accelerators programmable by non-hardware-experts. In light of this, such non-HLS frameworks, while useful as a means to inform on current paradigms and performance targets, cannot fall under the scope of this work (even in the rare instances where code is available).

Note that several frameworks, especially RTL ones, are quite loosely defined, and not at all user oriented – by which we mean the necessary resources and knowledge that would allow a

³See *e.g.*, [5] for a wider analysis, covering some works up to 2019.

programmer to use them is not readily available. This may indicate that such frameworks are not ready for “plug-and-play” usage, and may require quite specialised knowledge to operate.

We have marked the “Framework usage” column with italic text whenever a document is especially unclear on usability; in such cases, the description refers to the *minimum* work we have been able to ascertain a user would need to generate an accelerator. Some literature hints at further work required: for instance, [113] contains a single sentence regarding usability, stating: “Users can implement customized Accelerator Function Unit (AFU) on FPGA [...]”. This is where the user-defined kernels reside, indicating that additional work may be required to generate the AFU besides writing the kernels.

Much of the literature (including HLS work) is written mainly from a hardware engineer’s perspective. This may be indicative of the status of graph analysis in the FPGA world: it is a “hard” (unfriendly) problem, still requiring a fair amount of hardware knowledge and time investment to tackle, even when using HLS. Due to this, adoption of software-first paradigms is slow (an issue not exclusive to graph analysis). We base our argument for high-level tooling in this fact. Our argument is that, while HLS provides an adequate abstraction for many application domains (albeit a leaky abstraction, as looking at highly optimised HLS code quickly indicates), it simply is not sufficient in applications such as graph analysis – either with current compilation techniques or current FPGA card technology. As a result, the development and usage of frameworks and other tools may be *required* in order to move the burden of deep hardware knowledge away from the domain expert and allow FPGAs to penetrate these specific markets. Nonetheless, hardware-focused literature may prove useful in guiding attempts at a specific HLS graph algorithm implementation.

In general, the edge-centric streaming model is quite intuitive: edge data streams into the accelerator, which responds with results. This, coupled with the fact that this model maps naturally to a streaming/dataflow paradigm, which is favoured for FPGAs, indicates that it is a good target for handwritten (HLS) implementations of graph algorithms, especially by non-hardware-experts⁴; do note, however, that *formulating* an edge-centric algorithm may be non-trivial.

Regarding memory, a reoccurring issue is low global memory bandwidth. Many works assume that the graphs fit in BRAM in order to increase performance, or go to heavy lengths to ensure BRAM is optimally used even when handling large graphs, via buffering and caching mechanisms. Works which assume HBM – or a predecessor, Hybrid Memory Cube (HMC) – tend to focus less on bandwidth issues, indicating that high-bandwidth technologies might be a solution to these issues.

ThunderGP emerges as the most promising of the HLS frameworks. It is code-available, presents state-of-the-art results, and clearly specifies how it is used. This framework follows the GAS model. We have mentioned, in Section 2.6, that the GAS model restricts the types of computations that may be effectively expressed. In fact, the GAS model in itself constitutes a trade-off between generality and performance, especially in power-law graphs [23].

⁴Assuming the algorithm can be implemented satisfactorily following this model: streaming tends to require multiple passes for certain problems [5].

The GAS Model		
1:	while not done do	
2:	for all e in Edges do	▷ The Scatter stage
3:	$u = \text{new update}$	
4:	$u.dst = e.dst$	
5:	$u.value = \text{Scatter}(e.w, e.src.value)$	
6:	end for	
7:	for all u in Updates do	▷ The Gather stage
8:	$u.dst.accum = \text{Gather}(u.dst.accum, u.value)$	
9:	end for	
10:	for all v in Vertices do	▷ The Apply stage
11:	$\text{Apply}(v.accum, v.value)$	
12:	end for	
13:	end while	

Figure 3.1: The ThunderGP Gather-Apply-Scatter Formulation

(Source: [12])

Figure 3.1 shows the specific ThunderGP formulation of the GAS model. Note that the formulation makes the scatter phase explicit, and views the gather phase occurring over an *update* (generated by scatter) rather than over an edge, as in [41]. An update, however, is equivalent to an edge when the scatter function is nullipotent (or even omitted).

In contrast to academic work, Xilinx’s own Vitis Library offering spans a wider array of problems, but their application is largely untested save for the vendor’s own benchmarks versus CPU frameworks (see *e.g.*, the PageRank implementation [93]). Documentation is adequate (albeit lacking in some respects), and as usage is done directly at the HLS development level, workflows are simplified versus the usage of more complex frameworks.

We do note that vendor-provided libraries generally follow a vastly different development philosophy than that of academic work. While the libraries are (for the most part) production-ready, their evaluation may be less rigorous than tools published through the traditional academic cycle.

Chapter 4

Topics on High-Level Synthesis and FPGA Accelerator Boards

We present selected topics on FPGA accelerator boards (specifically, the Xilinx Alveo series we use in this work), and FPGA HLS programming. We have chosen these topics in order to provide coverage of basic knowledge required to later approach some points in our implementations (Chapter 5) and evaluation (Chapter 6).

4.1 High-Level Synthesis

Vitis HLS is Xilinx’s High-Level Synthesis tool, which maps kernels written in C, C++, or OpenCL to HDL. The development flow is split into three stages: software emulation, hardware emulation, and hardware implementation. Hardware implementation can be a very slow process, where compilation times start at half an hour, and easily achieve hours or tens of hours. Therefore, validation of the approach itself often takes place during the emulation stages. During the hardware implementation stage, the programmer confirms performance estimates and possibly locates any remaining bugs not exposed during hardware emulation [6].

Although Vitis allows programming Xilinx FPGAs with C/C++, this is still a “leaky” abstraction. Hardware-related details are not necessarily hidden from the programmer and usage relies heavily on custom pragmas to guide the compiler – meaning bare-metal considerations will never fully exit the programmer’s mind, although the programming paradigm itself (such as dataflow) may help abstraction [6].

The most important concepts to understand are dataflow, unrolling, and pipelining. A brief explanation of each of these follows.

4.1.1 Dataflow

The dataflow model (*i.e.*, task-level parallelism of functions and loops, where data “flows” from one task to the next) is the preferred paradigm whenever possible [101].

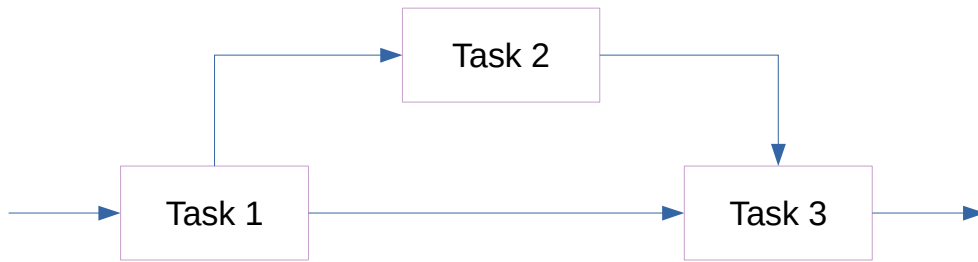


Figure 4.1: Dataflow: Example Dependencies

Dataflow allows a significant amount of parallelism to be extracted at low cost to the programmer. However, the large degree of underlying automation *enabling* programmers to simply add a dataflow pragma comes at a cost: within a dataflow region, a specific, strict coding style must be followed (termed dataflow canonical form), under penalty of poor performance or failure to compile. In addition, certain features should not be present, such as conditional execution of tasks (for instance, a conditional function call), or early loop exit [101]. Certain types of computations are more suited to the dataflow programming than others. Most importantly, streaming type computations can often be expressed as a dataflow.

For instance, an application consisting of three stages, with the interdependencies shown in Figure 4.1, will often be implemented sequentially in serial/naive CPU programming. This results in the suboptimal execution sequence shown in Figure 4.2. Applying the dataflow optimisation prompts the HLS compiler to analyse these dependencies and schedule each individual task to execute concurrently as soon as some data from its predecessors is available. This is shown in Figure 4.3. As can be seen, dataflow is an excellent way of enabling concurrency at low cost, for algorithms that support it.

Dataflow is enabled via the HLS `dataflow` pragma. An example is given in Listing 4.1 (note that the block scoping is unnecessary and just given for clarity).

```

1 #pragma HLS dataflow
2 {
3     task1(in, d2, d31);
4     task2(d2, d32);
5     task3(d31, d32, out);
6 }
  
```

Listing 4.1: Dataflow Example

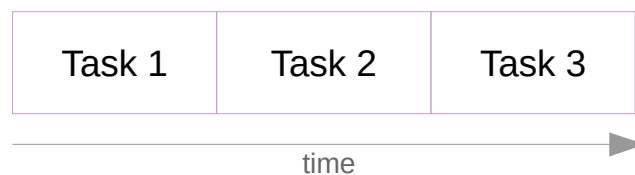


Figure 4.2: Dataflow: Example Serial Execution



Figure 4.3: Dataflow: Example Concurrent Execution

4.1.2 Unrolling

Loops are an essential construct in programming. Loops in CPU execution are often executed iteratively (one iteration following another) or blocked-iteratively (one block of iterations following another, where several threads concurrently execute one block of iterations if intra/inter-loop dependencies allow).

Loop unrolling is also a concept in CPU compilation, but loops optimised in this manner are still semantically sequential. Loop unrolling in HLS allows multiple copies of the loop to be generated in hardware, each of which executes fully concurrently.

If the number of loop iterations is known at compile time (for instance, processing a 256x256 bitmap), the loop may be *fully unrolled*: if we collapse the image-processing row-column loop, for instance, we may fully unroll the result and process all 65536 pixels concurrently (barring dependencies).

On the other hand, if the number of iterations is *not* known at compile time, we may generate n copies of the loop via *partial* loop unrolling. This still allows us to obtain concurrent performance.

Unrolling is performed via the HLS `unroll` pragma. An example is given in Listing 4.2.

```

1 for (int i = 0; i < 256*256; ++i) {
2 #pragma HLS unroll
3     uint16_t col = local_mem_img[i];
4     col &= 0xff0f // Set green channel to 0.
5     local_mem_img2[i] = col;
6 }
```

Listing 4.2: Unrolling Example: Image Processing

4.1.3 Pipelining

In instances where unrolling is not possible due to FPGA area usage constraints or loop dependencies, pipelining is likely a possibility. HLS pipelining similar to its equivalent in CPU design,



Figure 4.4: Pipelining: Example Serial Execution

and is analogous to dataflow for a loop or function body: whereas dataflow executes multiple *tasks* concurrently, after a dependency analysis, pipelining executes multiple *operations* concurrently.

An example is a read-accumulate loop on primary memory, as shown in Listing 4.3: in some instances, the loop cannot be unrolled, as doing so would require simultaneous accesses to the same memory bank. The non-pipelined loop will perform each of the two operations sequentially, as shown in Figure 4.4. Pipelining will allow the read operation to occur concurrently with the accumulate operation, as shown in Figure 4.5.

```

1 for (int i = 0; i < len; ++i) {
2   #ifdef DO_PIPE
3   #pragma HLS pipeline
4   #endif
5   const in = input[i];    // RD
6   acc += in;              // ACC
7 }

```

Listing 4.3: Pipelining Example: Read-Accumulate Loop

Pipelined loops require all their nested loops to be fully unrollable (otherwise, the operations cannot be scheduled properly, as their number is unknown at compile time). Pipelined loops can themselves be unrolled: this allows multiple pipeline iterations to execute in parallel.

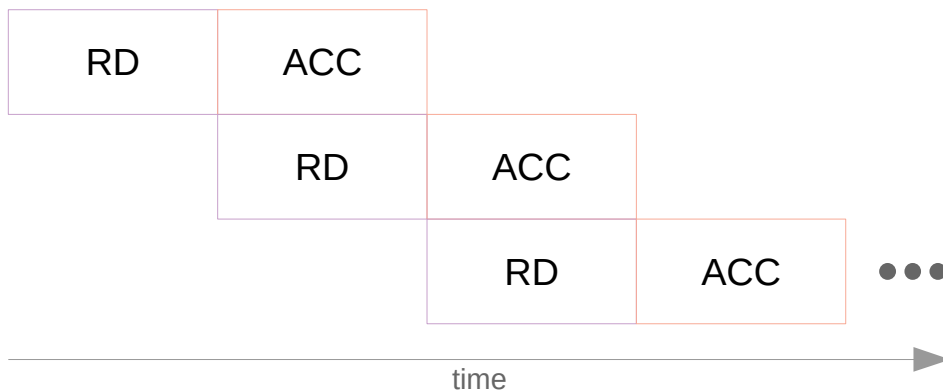


Figure 4.5: Pipelining: Example Pipelined Execution

4.2 FPGA Accelerator Boards

As mentioned, we will focus on the Xilinx Alveo series of FPGA Accelerator Boards. Some information will be specific to either the Alveo U250 or the Alveo U280, the two boards we use in our work.

4.2.1 Shells

Alveo boards are conceptually divided into two partitions. The user partition is programmed with the user-compiled binary and executes the application kernels. The shell partition provides system-wide, common infrastructure, such as PCIe and board management [99].

Each board must be first flashed with shell partition data. This data is then loaded onto the board at boot time, and allows additional firmware to be loaded after the OS boots [97]. Shell data may be revised or updated by the vendor. Throughout this document, we will use the shorthand “shell” to refer to shell versions.

In addition, some shells may further subdivide the original shell partition into *base* and *shell* partitions. This is the case for the `gen3x16_xdma_3_1` U250 shell. These are known as two-stage platforms. In this model, the base partition is loaded at boot time as before, but the shell partition must be explicitly loaded after boot. The shell partition may contain functions such as PCIe Direct Memory Access support.

A user binary is compiled against a specific shell version, so, in applications which require advanced functionality¹, the user must take care to select the correct shell.

4.2.2 Topology

In contrast, Alveo boards are *physically* divided into Super Logic Regions (SLRs). An SLR is a discrete die slice which is linked to other SLRs in the system via a silicon interconnect (Xilinx refers to this approach as “Stacked silicon interconnect”), easing the management of ever-increasing programmable fabric complexity [69]. The U250 contains 4 SLRs, while the U280 contains 3. Physically, memory banks connect to SLRs, and user logic must reside in the user region of the SLRs (kernels and shells both may span multiple SLRs) [100]. This is an additional consideration for the user as kernel placement may impact performance – for instance, if a kernel in SLR0 connects to a kernel in SLR3, or to a memory bank *tied* to SLR3, logic must be synthesised to create the link, entailing overhead). This is known as “kernel floorplanning” [89].

Figures 4.6, 4.7, and 4.8 give the topology for the U250 `201830_2`, U250 `gen3x16_base3`, and U280 `gen3x16_base3` platforms, respectively [96]. All blank space is to be considered user partition space.

Note that the U250 `gen3x16_base3` platform is two-staged, and holds less space in SLR1 for user kernels. Also note that, while the U280 contains two HBM stacks (each holding 4 GB), they are all tied to SLR0.

¹Such as QDMA, a DMA implementation which enables host-to-kernel streaming [90].

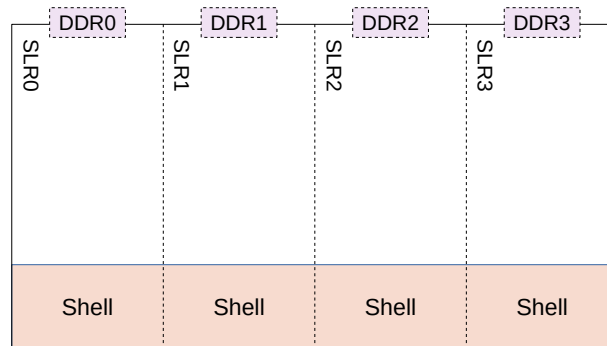


Figure 4.6: Topology for the Alveo U250 201830_2 Platform

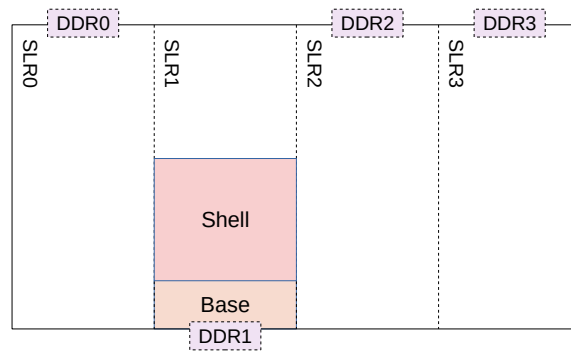


Figure 4.7: Topology for the Alveo U250 gen3x16_base3 Platform

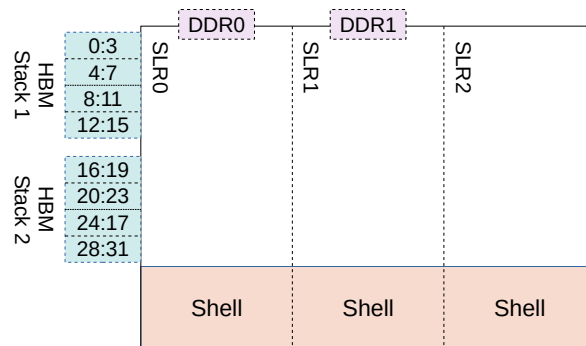


Figure 4.8: Topology for the Alveo U280 gen3x16_base3 Platform

4.2.3 HBM Considerations

One of the main advantages of HBM-based systems is increased bandwidth. This is achieved by packaging the processing unit and memory stacks together, thus eliminating the need for a longer external connection on the circuit board [86].

In addition, HBM provides many more memory channels. Each HBM stack provides 8 independently addressable memory channels, which can be further subdivided into 16 independently addressable “pseudo-channels” with half the bitwidth (from 128 to 64 bits) [78]. These pseudo-channels are often equated to memory banks, and are referred to as such in literature.

As the U280 contains two HBM banks, we have 32 concurrently accessible banks available to us, for a maximum theoretical bandwidth of 512 GB/s (documentation for the board gives a maximum of 460 GB/s [89]). Data-parallel kernels, for instance, may thus be replicated potentially many more times compared to DDR4-based accelerators, resulting in overall higher throughput.

The complex pseudo-channel and memory-to-logic infrastructure does, however, introduce some quirks which must be taken into account when using HBM-enabled boards. Some of the most significant are:

- Small write transactions spanning the two memory banks may suffer from overhead [89].
- Every four contiguous banks use common infrastructure, and are thus considered part of the same *bank group*. Consecutive accesses to the same bank group entail overhead [66].
- Using the entirety of the 32 available channels while executing more than one kernel may result in memory contention and corruption in one channel [89].
- Each bank group is serviced by a switch, which interfaces with the programmable fabric. A kernel port may span multiple bank groups – in order to support this, each switch connects to its neighbours laterally. Each two ports share one lateral connection, so maximum throughput in this instance is reduced by 50% [98].
- There is a latency cost when traversing switches [104].
- If multiple ports span the same two bank groups, the lateral connection may become saturated, reducing throughput [104].

Chapter 5

Implementation

As mentioned in Section 1.5, we have assembled a minimum set of algorithms which both: provide acceptable coverage of the graph analysis problem space; and have been mentioned as useful, or potentially useful, by our partners at IT4I. The resulting set is composed of the algorithms for: Single-Source Shortest Paths, PageRank, and Distinctiveness Centrality.

We have found no fully workable instance of a library or tool for SSSP or PR with Xilinx FPGA accelerator boards, save Xilinx’s own Vitis Graph Library¹. Our implementations of these algorithms make use of the kernels included in the library, with minor alterations in some instances.

5.1 Kernel Runner

The Kernel Runner (class `kernel_runner`) is our framework for executing and profiling kernels. While our kernels are written in Vitis C/C++ HLS, the host side uses the OpenCL API for device, program, queue, and kernel management. All kernels are scheduled using a single out-of-order queue. Internally, profiling data is obtained via a mixture of OpenCL event profiling (`clGetEventProfilingInfo()`), and calls to `std::chrono`.

The main loop for a Kernel Runner instance is given in Listing 5.1. As can be seen, the class is designed to be quite customisable via subclassing, and our applications make extensive use of this feature².

```
1 void run() {  
2  
3     setup_kernel();           // Virtual; setup kernel and kernel-related data  
4     setup_runs();            // Virtual; setup additional kernel inputs/outputs;  
                             setup/preprocess dataset  
5 }
```

¹We were unable to work with ThunderGP, likely due to outstanding bugs, but we still believe it has the potential to be a very useful tool; we reference this in Section 7.3.

²Nonetheless, as use cases evolved and specific implementations (especially DC1) required more complexity, subclasses contain a significant amount of logic that could potentially be refactored into the main class.

```

6   int n = 0;
7   for ( auto sourceID : sources ) {
8       time_basis = std::chrono::steady_clock::now();
9
10      /* Setup our FPGA energy reader at a sample frequency of 120 Hz */
11      energy_reader watts = energy_reader(time_basis, power_fn, 120);
12      if (profiling) watts.start();
13
14      std::this_thread::sleep_for(std::chrono::seconds(1));
15      run_kernel(sourceID);                // Virtual
16      post_run();                          // Virtual
17      std::this_thread::sleep_for(std::chrono::seconds(1));
18
19      if (profiling) {
20          watts.end();
21          profile_kernel();                // Virtual
22          do_json(sourceID, n, watts,      // Virtual
23                  std::chrono::time_point_cast<std::chrono::microseconds>(time_basis)
24                  .time_since_epoch().count());
25      }
26  }
27  at_end();                               // Virtual
28 }

```

Listing 5.1: Kernel Runner, Main Loop

We are able to obtain total FPGA card power draw via class `energy_reader` (which in turn accesses a sysfs file exposed by the card driver), and total *machine* power draw via a wall meter.

Since our use cases contain complex interactions between the host and the accelerator, we focused our study on machine power draw. However, the data we make available contains both metrics.

Application execution results and profiling data are dumped in JSON, via the JSON for Modern C++ library [52].

5.2 Single-Source Shortest Paths

Our SSSP implementation uses the Vitis Graph Library SSSP kernel [95], with no alterations. The kernel follows the common Bellman-Ford formulation, equipped with a FIFO queue to hold the current vertex frontier. This is, of course, a traversal algorithm, which requires constant random memory accesses to several data structures (as, for instance, vertices may connect to other vertices held in a much farther memory block). As mentioned in Section 2.1, this is an issue for current-gen FPGAs. Our analysis shows the effect of this characteristic.

Algorithm 1 gives the high-level plan followed by the Vitis Graph Library implementation (sourced from [95]).

Algorithm 1 High-Level Vitis Graph Library Bellman-Ford SSSP Algorithm

```

1: procedure SINGLESOURCESHORTESTPATH(graph, source)
2:   for each vertex  $v$  in graph do
3:     distance( $v$ )  $\leftarrow$  positive infinity
4:   end for
5:   distance(source)  $\leftarrow$  0
6:   while  $Q$  is not empty do
7:      $u \leftarrow \text{pop } Q$ 
8:     for each edge  $(u, v)$  in graph do
9:       if distance( $u$ ) + weight( $u, v$ ) < distance( $v$ ) then
10:        distance( $v$ )  $\leftarrow$  distance( $u$ ) + weight( $u, v$ )
11:        push  $v, Q$ 
12:       end if
13:     end for
14:   end while
15:   return distance
16: end procedure

```

We analyse only a single variant of SSSP (running on the Alveo U250), which we simply dub “SSSP”, as results for HBM execution given in [95] do not suggest this as a likely avenue for significant improvement.

5.3 PageRank

Our PR implementations use the Xilinx Graph Library PR kernel [93, 94], with minimal alterations to support additional inputs (damping factor and initial values).

The kernel follows the vertex-centric (non-algebraic) approach to PageRank. The formula followed for an iteration of PR is

$$R(v) = i_v + d \left(\frac{R(w)}{g_w^+} + \frac{R(x)}{g_x^+} + \dots \right), \quad (5.1)$$

where i_v is the initial value for v , d is the damping factor, and $w, x, \dots$ are vertices which connect to v .

The kernel itself is divided into three phases: weighted out-degree calculation (blocked, using local caches); buffer initialisation (which initialises PR values and the “constant” vector which holds the precalculated $(1 - \alpha) * \frac{1}{g_v^+}$ values; and the actual iterative stage (which is further subdivided into a blocked multiplication phase and a convergence check phase. Note that convergence is achieved when all updates for all vertices are less than some ϵ , rather than as a function of the entire vector (such as the norm of the difference between iterations).

Blocked phases make heavy use of local caches and ping-pong type buffers. The number/size of these buffers varies per implementation.

Table 5.1 shows the PR application variants we evaluated. Host-side code, save for FPGA management, is the same between variants.

Table 5.1: PageRank Application Variants

Name	Kernel Ver.	Notes
PR-DDR	pagerank	DDR implementation of the PageRank kernel.
PR-HBM2C	pagerank2c	HBM implementation of the PageRank kernel, using 2 channels for each buffer.
PR-HBM6C	pagerank6c	HBM implementation of the PageRank kernel, using 6 channels for each buffer.
PR-HBMJ	pagerankj	Equivalent to PR-HBM2C, but all buffers are at least mapped to double the original number of banks to support larger datasets.

5.4 Distinctiveness Centrality 1

Our Distinctiveness Centrality implementations receive more focus for several reasons: because the algorithm has several interesting properties (which we discuss in Section 7.2), because we have built it from the ground up, and because it is both a novel formulation of an algorithm and a novel implementation of an accelerator for said algorithm.

Out of five given DC metrics, only two (DC1 and DC5) require only information on each vertex’s direct neighbours, or information known at runtime via the most commonly used matrix representation data structures [27]. We have chosen the first as it is the most complex of the two.

We have selected the Compressed Sparse Row (CSR) representation (the very same as used in the SSSP implementation) as it allows the computation to be modelled in a fully data-sequential fashion, allowing both dataflow-style coding (the preferred style whenever possible, see Section 4.1), and scaling capabilities by dividing the dataset on vertex boundaries. A full exposition of this follows.

5.4.1 DC1 and its CSR formulation

Let $a(G)$ be the full $N \times N$ matrix describing graph G having a total of N vertices. Let w_{ij} be the weight of the edge connecting vertices i and j in G , or zero if no such edge exists. Let E be the total number of edges in G . Assume G is fully connected. Each entry in the diagonal is null, as traditional graphs do not allow self-edges:

$$a(G) = \begin{pmatrix} 0 & w_{12} & w_{13} \\ w_{21} & 0 & w_{23} \\ w_{31} & w_{32} & 0 \end{pmatrix} \quad (5.2)$$

We flatten the matrix column-wise and eliminate null entries via function f , obtaining the vector V :

$$V = f(a(G)) = (w_{12} \ w_{13} \ w_{21} \ w_{23} \ w_{31} \ w_{32}) \quad (5.3)$$

Flattening in this manner allows us to stream in edge data. However, we have lost all information regarding the shape of the matrix. We will eventually need to provide this data somehow.

The DC1 value for vertex i [27] is given by

$$D_1(i) = \sum_{j=1, j \neq i}^n w_{ij} \log_{10} \frac{N-1}{g_j^\alpha}, \quad (5.4)$$

where g_j is the degree of vertex j , and α is a tuning parameter.

Note that implementing Equation 5.4 requires *knowing* i , so as to attribute each instance of the sum to our selected vertex.

Let us assume an undirected graph for now ($w_{ij} = w_{ji}$ for all i, j). Swapping i for j and vice-versa throughout the equation except w_{ij} therefore gives the same result over G 's entire vertex set. We may then obtain the equation for D_1^0 , the DC1 value for an undirected graph:

$$D_1^0(j) = \sum_{i=1, i \neq j}^n w_{ij} \log_{10} \frac{N-1}{g_i^\alpha} \quad (5.5)$$

To avoid confusion, note that from this point on, whenever we refer to DC1, we mean the formulation given in Equation 5.5.

We now require knowing j (the destination vertex) instead of i . We therefore must supply this information by adding a new vector C which gives, for every entry in V , the destination of the corresponding edge:

$$C = (j : w_{ij} \in V) \quad (5.6)$$

Figure 5.1 shows V and C . These two vectors allow us to *partially* reconstitute the original matrix. We are, however, missing one final piece of information: we have no way of obtaining g_i ³.

V	$=$	w_{12}	w_{13}	w_{21}	w_{23}	w_{31}	w_{32}
C	$=$	2	3	1	3	1	2

Figure 5.1: Vectors V and C

We could, much like C , build a vector of $|E|$ elements giving the source of each entry in V – but if we note that the w_{ij} for every i are always contiguous and ordered (because we have flattened $a(G)$ column-wise), we may save storage space by simply storing the number of entries

³Note that we may obtain g_j by simply building a histogram over C , but this is neither efficient nor useful: the degrees of adjacent edges might very well be different, even in undirected graphs.

(i.e., degree) for each vertex as a vector— $D = (g_i : i > 0 \wedge i \leq N)$ — giving us direct access to a vertex's degree. We may now additionally fully reconstitute $a(G)$, if we so desire.

However, at this point, we are extremely close to the Compressed Sparse Row format. CSR is an extremely common data structure for sparse matrices and graphs, so in the interest of interoperability we will adjust our own data structures to match CSR (this is also the reason we swapped i and j in Equation 5.5).

CSR differs from our own data structure by instead storing the *index* of the first edge in V , for each node (w_{ij} minimising j), along with the first out-of-bounds index (which, conveniently, is equal to $|V|$ — the number of edges, or non-null entries, in G and $a(G)$ respectively). We name this vector O .

This difference in CSR optimises edge addressing (which we happen to not require for our DC1 formulation) at the lesser cost of increased computing time for obtaining $g_i = O(i+1) - O(i)$ (see Figure 5.3).

Figure 5.2 shows V , O , and C . We now switch to zero-indexing as we will soon move on to describing our implementation.

$(I) :$	0	1	2	3	4	5
$V :$	w_{01}	w_{02}	w_{10}	w_{12}	w_{20}	w_{21}
$C :$	1	2	0	2	0	1
$O :$	0	2	4	(6)		
$(S) :$	0	1	2	(3)		

Figure 5.2: Zero-Indexed Vectors V , C , and O , as well as Vector I (V Index) and S (C Index)

$(S) =$	0	1	2	(3)
$O =$	0	2	4	(6)
$(D) =$	$O(0+1) - O(0) = 2 \quad O(1+1) - O(1) = 2 \quad O(2+1) - O(2) = 2$			

Figure 5.3: Obtaining the Vector of Vertex Degrees, D , via C

5.4.2 DC1-CSR Algorithm

We now have all required data to calculate undirected DC1. We take CSR as input, so we term this formulation *DCI-CSR*.

As we have chosen to implement a dataflow kernel, we iterate sequentially over V , C , and O to compute each addend of the summation (recall that j is given by C). This means some other

process must collect the addends indexed to j and accumulate them *over* j , giving the final DC1 value for all $j \in [0, N - 1]$ – in essence, building a histogram.

To maximise compute, simplify kernel development, and better illustrate the particularities and challenges of HLS development, we have chosen to implement this function in the host machine. We will refer to the FPGA DC1 process as the *kernel*, and to the CPU accumulation process as the *histogram function*.

Algorithm 2 Distinctiveness Centrality 1 (Kernel)

Require: V and C are read-only bounded queues of length E .

Require: O is a read-only bounded queue of length $N + 1$.

Require: R_I and R_V are write-only bounded queues of length E .

```

1:
2: procedure DC1( $V, C, O, N, R_I, R_V, \alpha$ )
3:    $o_{-1} \leftarrow \text{pop } O$ 
4:    $n \leftarrow N - 1$ 
5:   for  $v \leftarrow 0$  to  $N$  do
6:      $o_v \leftarrow \text{pop } O$ 
7:      $g_v \leftarrow o_v - o_{v-1}$ 
8:      $t \leftarrow \log_{10} \frac{n}{g_i^\alpha}$ 
9:     for  $e \leftarrow 0$  to  $g_v$  do
10:       $i_{ve} \leftarrow \text{pop } C$ 
11:       $v_{ve} \leftarrow \text{pop } V$ 
12:      push  $i_{ve}, R_I$ 
13:      push  $v_{ve} * t, R_V$ 
14:    end for
15:  end for
16:  return
17: end procedure

```

Algorithm 3 Distinctiveness Centrality 1 (Histogram)

Require: R is an array of length N .

Require: R_I and R_V are read-only bounded queues of length E .

```

1:
2: function HIST( $R_I, R_V, E, R$ )
3:   for  $e \leftarrow 0$  to  $E$  do
4:      $i \leftarrow \text{pop } R_I$ 
5:      $v_i \leftarrow \text{pop } R_V$ 
6:      $R_i \leftarrow v_i$ 
7:   end for
8:   return  $R$ 
9: end function

```

Algorithm 2 shows our design for the FPGA kernel. We have modelled a streaming computation using read-only or write-only queues. Note particularly that queue R_I is in essence simply a replication of queue C . We are assuming in-order processing at the moment, but we return R_I nonetheless in order to prevent eventual task-level reordering, or allow histogram processing to be

offloaded to a different site. This comes at no cost to the performance of our baseline implementation (except an increase in kernel-to-host read times significant only in trivially small datasets).

Algorithm 3 gives our design for the histogram function executing on the host. Note that in the actual implementation, R_I and V_I are not queues but rather arrays, making the problem embarrassingly parallel (and we do indeed capitalise on this via OpenMP).

We will now drop our assumption that G is undirected. This invalidates our algorithm – it is implied to be universally applicable, but D_1^0 is not defined for vertices of a directed graph. Indeed, if we execute the algorithm, we are capturing only the inbound edges of each vertex.

$$D_1^-(j) = \sum_{i=1, i \neq j}^n w_{ij} \log_{10} \frac{N-1}{g_i^{+\alpha}} \quad (5.7)$$

Equation 5.7 defines $D_1^-(j)$, Distinctiveness Centrality 1 In (or in-DC1), where g_i^+ is the out-degree of vertex i (its number of outbound vertices). Note that, when inputting a CSR matrix corresponding to a directed graph, we are specifically capturing g_i^+ via O . We may therefore restate Algorithm 2 as giving the solution to D_1^0 for undirected graphs, and D_1^- for directed graphs.

A full formalisation begs clarification on the conversion from graph to matrix. Per our reasoning in Subsection 5.4.1, the algorithm is only valid for undirected graphs if the matrix is symmetric (see 5.2), *i.e.* there is a two-to-one mapping from matrix entries to edges. If the matrix is *not* symmetric, the result is the in-DC1 vector corresponding to a directed reinterpretation of the graph.

5.4.3 Baseline Kernel

Our baseline kernel is termed DDR1. This kernel follows Algorithm 2 closely. The main execution cycle is shown in Listing 5.2 (the full device source code is given in Appendix A.1). Note that the compute happens strictly in function `dcl_stream()`: the remaining code is initialisation logic for the streams as well as compiler directives.

The `execute` loop runs for every vertex; it is responsible for initialising the current vertex’s degree, and the t value given in the algorithm. The `execute_inner` loop runs for each edge of the current vertex; it calculates the edge’s R_V value and pushes this (along with R_I).

Note that `execute_inner` is pipelined (to an initiation interval of 1), while `execute` is not. As referenced in Section 4.1, pipelined loops must unroll their descendants, and `cur_deg` is a runtime parameter, so unrolling is impossible. This means that, while `execute_inner` is capable of outputting one result per cycle, we incur `execute`’s non-negligible latency cost *for every vertex*; and processing of a vertex’s edges cannot commence until this cost is paid.

A possible alternative would be to define a compile-time maximum number of edges for every vertex (*e.g.*, a maximum degree), allowing `execute_inner` to be fully unrolled (this would also require restructuring so pushing several values at once becomes possible). Defining maximum limits for dataset properties is common in HPC, and even more so in HLS⁴. We certainly do not claim this is an inadequate solution (indeed it is a likely necessity for optimal versions of some, if

⁴Refer to the Vitis Graph Library source [102] for several examples of this.

not all, graph applications), but we chose not to pursue this path as it limits flexibility. Additionally, as we will detail in Subsection 5.4.5.3, this particular point is no longer a concern as we advance in our kernel’s optimisation.

```

1  template <int ALPHA>
2  inline void dcl_stream( hls::stream<unsigned> &offset_fifo,
3                        hls::stream<unsigned> &column_fifo,
4                        hls::stream<unsigned> &weight_fifo,
5                        hls::stream<unsigned> &out_idx_fifo,
6                        hls::stream<float> &out_val_fifo,
7                        const unsigned num_vertices) {
8
9
10 setup:
11     static unsigned prev_offset = offset_fifo.read();
12     static const unsigned numer = num_vertices-1;
13
14 execute:
15     for (unsigned j = 0; j < num_vertices; ++j) {
16         const unsigned cur_offset = offset_fifo.read();
17         const unsigned cur_deg = cur_offset - prev_offset;
18
19         const float cur_deg_pow = (float) std::pow(cur_deg, ALPHA);
20         const float tree = std::log10(numer / cur_deg_pow);
21
22
23 execute_inner:
24     for (unsigned edge = 0; edge < cur_deg; ++edge) {
25 #pragma HLS PIPELINE II=1
26
27         const unsigned weight_tmp = weight_fifo.read();
28         const unsigned i_tmp = column_fifo.read();
29
30         const float value_tmp = tree * weight_tmp;
31
32         out_idx_fifo.write(i_tmp);
33         out_val_fifo.write(value_tmp);
34
35     }
36     prev_offset = cur_offset;
37 }
38 return;
39 }

```

Listing 5.2: DC1 Kernel (Baseline, Execution Cycle)

5.4.4 Variants

Our baseline has, over time, been augmented with optimisations, whether intrinsic or extrinsic to the code itself. Intrinsic optimisations include, for instance, casting the parameters of `std::pow` to float, thus preventing the compiler from inferring the `std::pow(double, double)` version, which is slower, and not the `std::pow(float, float)` version⁵. Extrinsic optimisations include increasing the number of compute units.

Additionally, we have explored both a DDR4-based board and a HBM-based board (see Section 4.2 and Section 6.2), so each optimisation may correspond to up to two variants.

Table 5.2 shows the correspondence between our DC1 application variants, their applied optimisations (see Subsection 5.4.5), and the FPGA kernel name, as well as a brief explanation of each variant. Note that this in no way an exhaustive list of all explored variants: several other optimisations were attempted which, due to several factors (including, of course, not actually providing any improvement) were excluded from our evaluation.

As we will explain in Subsection 5.4.5, our more advanced variants occasionally no longer require some of the input or output queues. Whenever possible, we have *kept* these queues regardless, in order to focus on compute-level optimisations rather than pollute the results with altered host-device transfer times. Our final optimisations do not include any unnecessary transfers, however.

5.4.5 Optimisations

5.4.5.1 J

The J optimisation entails mapping a kernel port to more than one bank, allowing larger datasets. The argument means the maximum number of banks assigned to one port. This is a purely extrinsic optimisation, requiring no code changes. Note that we restrict ourselves to placing compute units in the same SLR as *all* the memory banks they access. This restricts our Alveo U280 implementations to SLR0 as per the topology given in Subsection 4.2.2.

In practice, several variants of the optimisation are possible, but we have chosen to set *O* at two ports and maximise the remaining queues of identical size at seven ports each.

5.4.5.2 S

The S optimisation entails partitioning the dataset into n blocks, each of which is executed in parallel by a compute unit. This requires both altering the kernel to distinguish between the *total* number of vertexes (necessary to obtain n) and the number of vertexes in the current partition (N).

Additionally, it is necessary to alter the host-side logic to support multiple compute units, and perform the partitioning in the first place. Listing 5.3 gives our code for graph partitioning. In essence, we view the edge as a primitive object⁶ and divide our dataset between the CUs so that

⁵This occurs because integer arguments to `std::pow()` are promoted to `double`.

⁶In other words, the ultimate target of computation.

each is assigned roughly the same number of edges. We specify “roughly” because we cannot partition the edges fully equally: our application assumes, for the sake of managing complexity, that a vertex processing step *always* has that vertex’s full edge set available. This means that our edge partitioning must respect vertex boundaries.

In theory, assuming a low variance in vertex degree, division by vertex rather than by edge would suffice. On the other hand, if there exists a significant amount of degree variance, but the average degree is low enough to not overwhelm the high cost of processing a vertex in our baseline, division by vertex may also be more equitable. However, this ceases to be the case when the D optimisation is applied.

```

1 void setup_buffers() {
2
3     /*
4         cu: number of compute units
5         d: graph struct
6         v: full vertex set (offset array)
7         kd: kernel data struct array, where buffers for
8             CUs are kept
9     */
10
11     unsigned n = d.num_edges;
12     unsigned bs = n/cu;    // block size
13     long new_edge, cur_edge = -1;
14     long new_vertex, cur_vertex = -1;
15
16     auto s = std::begin(v);
17     auto b = std::begin(v);
18     auto e = std::end(v);
19
20
21     // We want to divide roughly by edges, since this is where the compute happens
22     // But to keep our implementation simple we cannot assign *parts* of a vertex's
23     // edge set,
24     // but rather all our vertexes must have their full edge sets included
25     for ( unsigned k = 0; k < cu; ++k ) {
26         KernelData &kd = kernel_data[k];
27
28         if ( cur_edge == d.num_edges ) {
29             // No more edges, all remaining CUs will do nothing
30             new_edge = cur_edge;
31             new_vertex = cur_vertex;
32
33         } else if ( k < cu-1 ) {
34             new_edge = bs*(k+1);    // "Naive" edge partitioning
35
36             // Find vertex with this edge
37             // Vertex 0 is given by 0-1, so we say vertex 0 is at 1

```

```

37     // and subtract accordingly
38     new_vertex = std::lower_bound(b, e, new_edge)-s - 1;
39
40     // Now adjust partitioning so we get *all* edges of this vertex
41     // (given by the index at v+1, which is the first edge of v+1)
42     new_edge = v.at(new_vertex+1) - 1;
43
44     } else {
45         // Last CU, just get all remaining elements
46         new_vertex = d.num_vertices-1;
47         new_edge = d.num_edges-1;
48     }
49
50     // Advance buffer
51     kd.column_offset = cur_edge + 1;
52     kd.offset_offset = cur_vertex + 1;
53
54     kd.column_size = new_edge - kd.column_offset + 1;
55     kd.offset_size = new_vertex+1 - kd.offset_offset + 1;
56
57     cur_edge = new_edge;
58     cur_vertex = new_vertex;
59 }
60 /* ... */
61 }

```

Listing 5.3: Graph Partition Logic for Multiple CUs

5.4.5.3 D

The D optimisation is a purely intrinsic optimisation – but it is leveraged to great effect, as we will show in Subsection 6.3.3. We reshape the `execute` loop to work on *edges* rather than *vertices*, so we move from an vertex-centric to an edge-centric paradigm. This is non-trivial because there exists a one-to-many relationship between the degree of the current vertex and its edges, and the edge processing stage must know what its current degree is. A singular edge-processing loop requires some additional logic, and thus complexity, to know the current degree for each iteration.

A possible solution could be, for instance, to introduce a counter variable which is initialised with the current degree and decremented each edge iteration; reaching 0, the counter would then trigger an update to the current degree and its own reinitialisation. However, adding such a variable introduces an inter-iteration dependency which may hamper or prevent further optimisations (such as unrolling or pipelining).

Our solution is one that is likely somewhat unique to FPGA programming: we preprocess the incoming vertex/offset stream to produce a degree stream (so that, for instance, (0,1,4) is converted to (1,3,3,3)). In essence, we are decoupling the process of calculating the current degree from the process of calculating t , which is much more costly. This allows both to pay a much lesser

cost per vertex (merely updating the current degree), and to pay this cost concurrently with edge processing; but, most importantly, it removes the nested loop structure of `execute` and allows us to fully pipeline the loop, thus coming much closer to the maximum of one output per cycle. Algorithm 4 shows the scheme for this optimisation.

Algorithm 4 Distinctiveness Centrality 1 with Optimisation D

Require: D is a read-write bounded queue of length E .

Require: O is a read-only bounded queue of length $N + 1$.

Require: V and C are read-only bounded queues of length E .

Require: R_V is a write-only bounded queue of length E .

```

1:
2: procedure DC1( $D, V, C, O, N, R_V, \alpha$ )
3:   do concurrently
4:     PROCDEGREE( $O, N, E, D$ )
5:     DC1DEGREE( $D, V, C, N, R_V, \alpha$ )
6:   end
7: end procedure
8:
9: procedure PROCDEGREE( $O, N, E, D$ )
10:   $g_{-1} \leftarrow \text{pop } O$ 
11:  for  $v \leftarrow 0$  to  $N$  do
12:     $o_v \leftarrow \text{pop } O$ 
13:     $g_v \leftarrow o_v - o_{v-1}$ 
14:    for  $i \leftarrow 0$  to  $g_v$  do
15:      push  $g_v, D$ 
16:    end for
17:  end for
18: end procedure
19:
20: procedure DC1DEGREE( $D, V, C, N, R_V, \alpha$ )
21:   $n \leftarrow N - 1$ 
22:  for  $e \leftarrow 0$  to  $E$  do
23:     $g_v \leftarrow \text{pop } D$ 
24:     $t \leftarrow \log_{10} \frac{n}{g_i^\alpha}$ 
25:  end for
26: end procedure

```

5.4.5.4 P

The P optimisation is quite simple: noticing that α is known at compile time, as per our baseline kernel, we simply substitute the call to `std::pow` with an unrolled multiply loop. We only apply this optimisation in conjunction with optimisation S, which only pays the latency cost once, so its usefulness is limited – although we found this necessary for reasons explained in Subsection 6.3.3.2.

5.4.5.5 C

The C optimisation is a host-side alteration for multi-CU variants. Previously, each kernel corresponding to a CU was enqueued sequentially (although the kernels themselves may run concurrently on the device), and after synchronising to all kernels, the accumulation stage is performed.

The C optimisation divides each block into *chunks*. A single compute unit/kernel may be assigned one or more chunks. Our analysis only explores the single-chunk-per-CU instance, but an exploration of the effects of altering this parameter is given in Appendix A. The number of chunks per CU is given as *CBx*, so we mostly refer to this optimisation as CB1.

The enqueue loop is reshaped to iterate over chunks, rather than CUs, and is additionally parallelised. Each chunk iteration then performs the accumulation over its own results: in essence, interleaving the accumulate stage with the device execution stage, leaving the CPU idle during a lesser amount of time and improving total execution time. Additionally, thread creation overhead is reduced, as there is now a single parallel loop (with a nested parallel accumulate loop). Listing 5.4 gives the host-side code for this optimisation.

```

1 void run_kernel(unsigned int sourceID) {
2     // sourceID is unused
3
4     setup_buffers();
5
6     unsigned hke = 0; // Host kernel enqueue: device time with host time
7     unsigned hre = 0; // Host read end
8
9     omp_set_nested(1);
10    std::chrono::nanoseconds accumulate_start =
11    std::chrono::steady_clock::now() - time_basis;
12
13    #pragma omp parallel for num_threads(chunks) default(shared)
14    for ( unsigned c = 0; c < chunks; ++c ) {
15        auto &cur_kd = kernel_data[c]; // Data for current chunk
16        auto &kernel = cur_kd.kernel;
17
18        const auto q_c = q.get(); // C API OpenCL queue
19
20        unsigned update;
21
22        unsigned arg = 0;
23
24        // kernel setup is not thread safe
25        omp_set_lock(cur_kd.kernel_lock);
26
27        kernel.setArg(arg++, cur_kd.offset_buf);
28        if(use_col) kernel.setArg(arg++, cur_kd.column_buf);
29        kernel.setArg(arg++, cur_kd.weight_buf);
30        if(use_idx) kernel.setArg(arg++, cur_kd.result_idx_buf);

```

```

31     kernel.setArg(arg++, cur_kd.result_val_buf);
32     kernel.setArg(arg++, (unsigned) d.num_vertices);
33     kernel.setArg(arg++, cur_kd.offset_size-1);
34     kernel.setArg(arg++, cur_kd.column_size);
35
36
37     clEnqueueMigrateMemObjects(q_c, cur_kd.ob_in_n, cur_kd.ob_in, 0, 0, nullptr,
38                               &cur_kd.e_write); // Write
39
40     clEnqueueTask(q_c, kernel.get(), 1, &cur_kd.e_write, &cur_kd.e_kernel); //
41     Enqueue
42
43 #pragma omp atomic capture
44     update = ++hke;
45     if (update == 1) {
46         host_kernel_enqueue = std::chrono::steady_clock::now() - time_basis;
47     }
48
49     omp_unset_lock(cur_kd.kernel_lock);
50
51     clEnqueueMigrateMemObjects(q.get(), cur_kd.ob_out_n, cur_kd.ob_out, 1, 1, &
52                               cur_kd.e_kernel, &cur_kd.e_read); // Read
53
54     clWaitForEvents(1, &cur_kd.e_read); // Wait for this chunk to terminate
55
56
57 #pragma omp atomic capture
58     update = ++hre;
59     if (update == chunks) {
60         host_read_end = std::chrono::steady_clock::now() - time_basis;
61     }
62
63     // Accumulate
64     {
65         const __restrict auto &col = cur_kd.column;
66         const __restrict auto &val = cur_kd.result_val;
67         const __restrict auto &res = result;
68
69         const auto sz = cur_kd.column_size;
70
71 #pragma omp parallel for simd schedule(static)
72         for (unsigned i = 0; i < sz; ++i) {
73             const unsigned idx = col[i];
74 #pragma omp atomic update
75             res[idx] = res[idx] + val[i];
76
77         }}
78     }
79
80     std::chrono::nanoseconds accumulate_end = std::chrono::steady_clock::now() -
81     time_basis;

```

```
76     q.flush();  
77     q.finish();  
78  
79 }
```

Listing 5.4: Run Kernel Function for C Optimisation

5.4.5.6 U

The U optimisation sets the port transfer size to 512 bits, and concatenates every 16 (32-bit) values into a single element. This change allows us to fully unroll the kernel execution loop in order to process 16 elements at a time. Doing so requires the D optimisation, as we must have a single edge-centric loop.

However, we must also generate a 512-bit element queue for our degree data. This is non-trivial to implement: we are processing 16 vertices at a time, but have no indication of the number of edges in that vertex set, thus making unrolling a difficult problem (at worse, we could have 16 vertices with no outbound edges, leaving us with a cycle where no degree data is output). Thus, this degree queue generation step is performed on the host. To improve performance, the U optimisation is only applied when the C optimisation is also applied. This introduces another nested parallel loop, *before* kernel enqueueing, where the degree queue is generated. [Appendix A.6](#) gives the full source code for the device kernel.

Table 5.2: Correspondence between DC1 Versions, Optimisations, and Kernel Versions

Name	Opt.	Kernel Ver.	Notes
DDR1	N/A	dc1	The baseline.
HBM1J7	J(7)	dc1	The baseline kernel compiled to an HBM board; "J7" denotes that each kernel parameter takes at most 7 HBM banks.
DDR4	S(4)	dc1_s	The baseline kernel and overall application adapted to support partitioning of the dataset and mapping onto 4 compute units.
HBM8	S(8)	dc1_no_idx	Equivalent to DDR4 on an HBM board using 8 CUs, except with no R_I queue.
DDR4OD	S(4), D	dc1_deg	The dc1_no_idx kernel adapted to use the degree pre-computation optimisation.
DDR4ODP	S(4), D, P	dc1_deg_pow	The dc1_deg kernel adapted to use the pow unroll optimisation.
HBM8OD	S(8), D	dc1_deg	HBM equivalent of DDR4OD.
HBM8ODP	S(8), D, P	dc1_deg_pow	HBM equivalent of DDR4ODP.
DDR4ODPCB1	S(4), D, P, C(1)	dc1_deg_pow	Chunk-based host side parallel kernel enqueueing
DDR4ODPUCB1	S(4), D, P, C(1), U	dc1_unroll	Like dc1_no_idx, but the vestigial column queue is replaced by a vertex degree queue, and the kernel loops are fully unrolled to a factor of 16.
HBM10ODP	S(10), D, P	dc1_no_col	Like dc1_no_idx, but the vestigial column queue is removed to make room for 10 CUs.
HBM10ODPCB1	S(10), D, P, C(1)	dc1_no_col	HBM equivalent of DDR4ODPCB1.
HBM10ODPUCB1	S(10), D, P, C(1), U	dc1_unroll	HBM equivalent of DDR4ODPUCB1.

Chapter 6

Evaluation

We present our results for SSSP, PR, and DC1 execution. We will also discuss the causes and possible outcomes of these results.

6.1 Metrics

Two kinds of metrics were collected: time- and energy-efficiency based. In particular, the relevant time-efficiency metrics are:

- Kernel execution time (SSSP, PR, DC1)
- Total execution time (DC1)

Energy-efficiency data was collected solely in one instance (see Subsection [6.2.1](#)):

- Write+kernel+read (“device”) energy consumption (DC1)
- Accumulate energy consumption (DC1)
- Total energy consumption (DC1)

When evaluating DC1, total execution time is obtained by summing total device time (the end time of the last host-to-device read, minus the start time of the first host-to-device write) with accumulate time (time spent by the host performing the histogram function). Similarly, total power draw is obtained as average energy consumption during device execution times total device time, plus average energy consumption during accumulate time times accumulate time.

6.2 Experimental Setup

6.2.1 Alveo U250: Local Machine

For applications using the Alveo U250, a local PC (dubbed `local`) was used. This PC has the following specifications:

- ASRock TRX40 Creator Motherboard

- 32 GB DDR4 RAM
- AMD Ryzen Threadripper 3960X CPU
- NVIDIA GeForce GTX 1650 Super GPU

When evaluating SSSP or PR the machine was used as-is. However, when evaluating DC1, the graphics card was removed and the operating system was executed at runlevel 3. Idle power draw for the system in this state is approximately 113 W.

Power consumption was measured using a UT230B power meter, which connects directly to the wall socket, in series with the machine's power supply. Accuracy for power measurement is $\pm 1\%$. This device has no data-logging feature, and sample frequency is approximately one second, so results should be taken as indicative only. In variants with sequential device (write/kernel/read) and accumulate stages, we record maximum power draw for each stage a minimum of 8 times, remove any outliers, and give the actual power draw as the average of the remaining results. In variants where accumulation is interleaved with kernel execution, we record power draw as above, but for the single write/kernel/read/accumulate stage.

The machine is equipped with the following software:

- Ubuntu 18.04 LTS (Linux kernel major version 4)
- Xilinx Runtime Library (XRT) 2.9.317
- Xilinx Vitis 2020.2
- GCC 7.5.0
- GCC 11.1.0

6.2.2 Alveo U280: Remote Machine

For applications using the Alveo U280, a remote virtual machine (dubbed `remote`) was used. The virtual machine is one of three executing on the same node. The physical machine has the following specifications:

- 376 GiB DDR4 RAM
- 4x Intel Xeon Gold 6234

The virtual machine is equipped with the following software:

- Ubuntu 18.04 LTS (Linux kernel major version 4)
- Xilinx Runtime Library (XRT) 2.5.309

6.2.3 Compilation

All host-side code was compiled using `local`.

All device-side code, with the exception of non-baseline DC1 variants, was compiled using `local`. These variants were compiled on a second local machine using Xilinx Vitis 2021.1.

Table 6.1: Dataset Shorthands and Origin

Shorthand	Full Name	Source
SO	soc-orkut	[67]
SO ^T	soc-orkut (transposed)	[67]
LJ	soc-LiveJournal1	[49]
USA	USA-road-d.USA	[21]
RMAT	RMAT22	[109]
R4	r4-2e23	[109]
CO	com-Orkut	[49]

All Alveo U250 code targeted the XDMA `gen3x16_base3` shell, with the exception of PR variants, all of which were compiled for the XDMA `201830_2` shell. All Alveo U280 code targeted the XDMA `201920_3` shell.

All device-side code was compiled with `-O3`. C and U variants were compiled with GCC 11.1.0 to take advantage of OpenMP 5.0 features; all other variants were compiled with GCC 7.5.0.

6.2.4 Datasets, Parameters, and Comparison Benchmarks

We evaluate the SSSP and PR implementation versus previously published benchmarks pertaining to CPU or GPU execution. GPU results are sourced from [7], which we shorthand as *Graphit*; CPU results are sourced from [109], shorthanded as *Zheng*.

DC1 is evaluated versus our own serial C++ implementation of an application following the same general lines as the DC1 baseline (with the exception that accumulation is done inline and not as a separate stage).

Table 6.1 shows the input datasets, their corresponding shorthands, and their source. Table 6.2 shows the algorithms and source for performance data we use for each dataset (where “*this*” indicates our own DC1 C++ results), along with relevant metadata. $|E|$ means “number of edges”; $|V|$ means “number of vertices”; *Dens.* gives graph density as calculated by $\frac{|E|}{|V|(|V|-1)}$; *Sym.* indicates whether the graph/matrix is symmetric: the edge counts given are the actual edge counts present in the matrix representation of the dataset.

When weighted graphs were required to execute SSSP, we used the random weights generated by the authors. In accordance with sourced data, all SSSP tests were executed using the first vertex as the source, and all PR tests use the same epsilon value and maximum iteration count. All DC1 tests were executed with $\alpha = 5$.

Table 6.2: Author-Algorithm-Dataset Correspondence

Author	Algorithm	Dataset	$ E $	$ V $	$\frac{ E }{ V }$	Dens.	Sym.
Graphit [7], (<i>this</i>)	SSSP, DC1	SO ^T	106M	3.00M	35.48	1.18e-05	No
Graphit	PR	SO	213M	3.00M	70.97	2.37e-05	Yes
Graphit, (<i>this</i>)	SSSP, DC1	LJ	68.4M	4.85M	14.11	2.91e-06	No
Graphit	PR	LJ	85.6M	4.85M	17.66	3.64e-06	Yes
Zheng [109], (<i>this</i>)	SSSP, PR, DC1	USA	57.7M	24.0M	2.41	1.01e-07	No
Zheng, (<i>this</i>)	SSSP, PR, DC1	RMAT	33.0M	4.19M	7.86	1.88e-06	No
Zheng, (<i>this</i>)	SSSP, PR, DC1	R4	34.0M	8.39M	4.00	4.77e-07	No
(<i>this</i>)	DC1	CO	234M	3.07M	76.28	2.48e-05	Yes

6.3 Results

Time-performance is displayed as the relative speedup of the CPU/GPU implementation over the FPGA implementation, except for DC1, where we show FPGA over CPU results. We do this to ease comprehension by displaying integers greater than zero rather than mostly reals in $]0, 1]$. DC1 Energy-performance is given as the absolute Watt-second value.

For SSSP and PR, we compare against a number of CPU/GPU frameworks (previously introduced in Section 2.6; Gunrock, GSWITCH, and SEP-Graph are given the shorthands GU, GSW, and SEP-G, respectively), or “Serial” for serial CPU execution – see the cited papers for more details.

As a rule of thumb, whenever results for more than one machine are given, we compare against the most time-performant. Additionally, both [7] and [109] give results for their own state-of-the-art frameworks. Although they are, for the most part, better performing versus the data we compare against, we have omitted these as well. Our focus is on comparing against currently-in-use, *established* frameworks, thus excluding these untested novel frameworks due to our usability condition mentioned in 3.

Absolute execution metrics for all implementations are given in Appendix B.

Note that in figures for SSSP and DC1, SO^T is labelled simply as *SO*.

6.3.1 SSSP versus High-Performance Frameworks

Figure 6.1 shows FPGA SSSP time-performance versus [109]’s benchmarks of CPU frameworks (as well as unoptimised serial execution). We were unable to execute SSSP using the USA dataset after five hours – this may be due to a conjunction of the kernel’s internal use of a queue, FPGA-unfriendliness, and the graph’s extreme sparsity or number of vertices – so it is excluded from our comparisons. Note that the speedup for R4 is slowed down approximately two times

versus RMAT, which corresponds somewhat closely to the ratio between vertex counts for the two datasets, again likely indicating the SSSP kernel’s suboptimal handling of vertex counts.

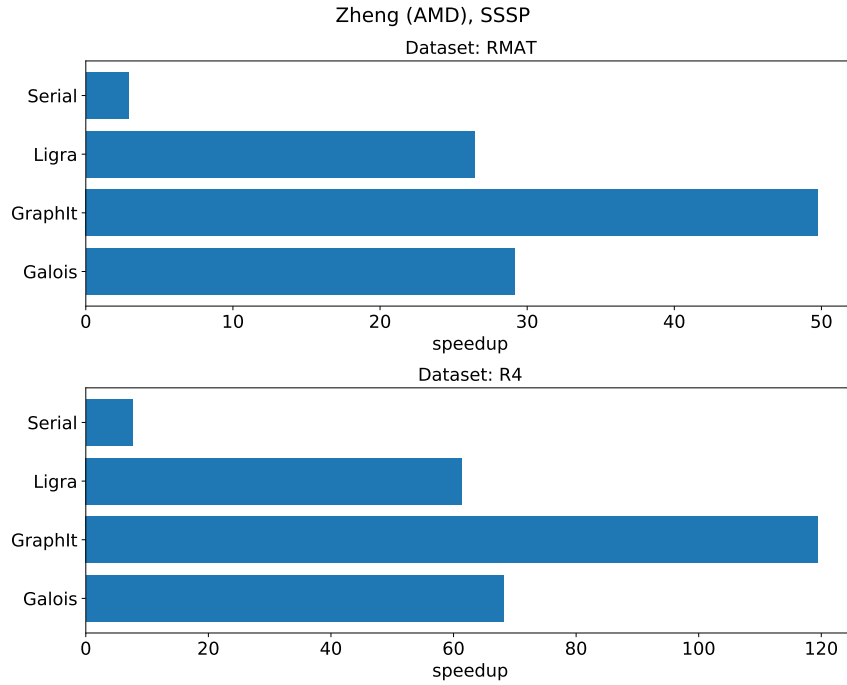


Figure 6.1: Execution Time for SSSP – CPU (Zheng/AMD) over FPGA, per Dataset and Framework

Figure 6.2 paints a similar picture – note that SO^T is less sparse than LJ, and has a higher edge-to-vertex ratio – the GPU frameworks also seem to be affected by the issue. Nevertheless, GPU speedups in this instance are, of course, much higher.

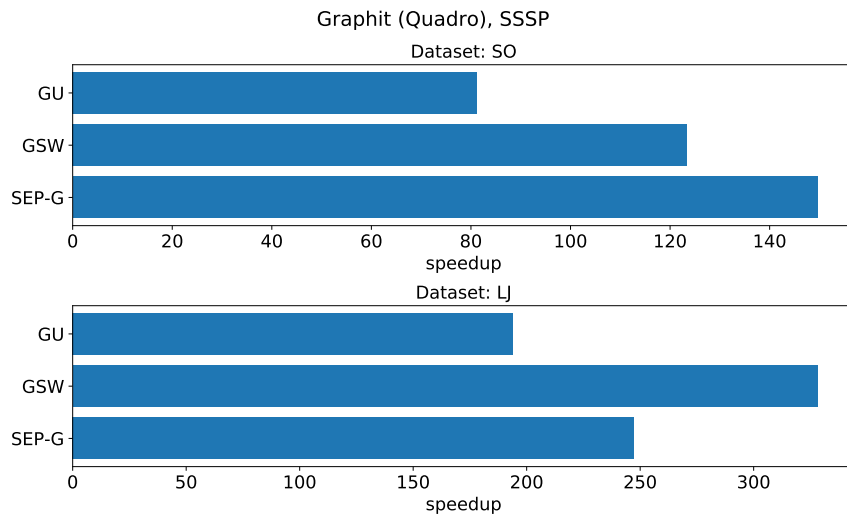


Figure 6.2: Execution Time for SSSP – GPU (GraphIt/Quadro) over FPGA, per Dataset and Framework

6.3.2 PageRank versus High-Performance Frameworks

6.3.2.1 Alveo U250

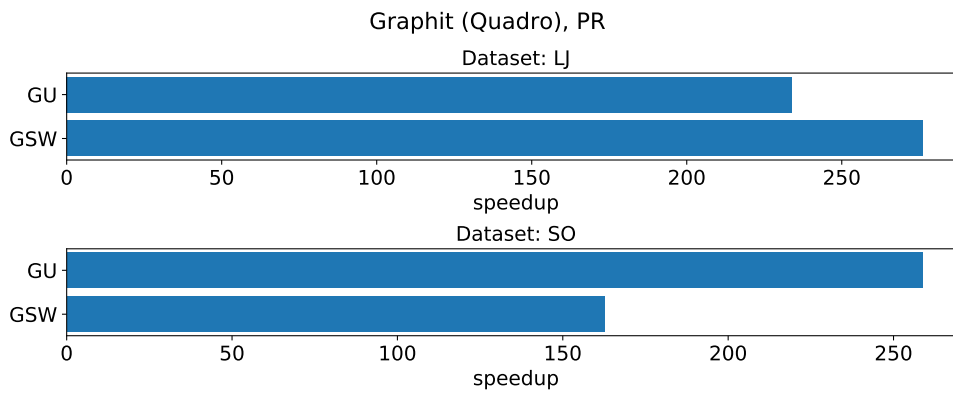


Figure 6.3: Execution Time for U250 PR – GPU (Graphit/Quadro) over FPGA, per Dataset and Framework

Figures 6.4 and 6.5 display results for FPGA PR-DDR vs. CPU PR execution. Versus the Intel machine, the FPGA implementation bests serial execution *and* the Ligra framework version for USA, and slowdowns versus the other frameworks are quite small. In fact, the equivalent data for the AMD machine is much the same: only serial execution is slower, but slowdowns versus the graph frameworks are still quite manageable (possibly excepting Galois). One factor which will be repeated is that results versus CPU appear to be overall ordered by sparsity: it seems CPU frameworks favour denser matrices when compared to FPGA execution.

Less encouraging are the results versus the highly optimised GPU frameworks shown in Figure 6.3. Again, it seems these frameworks handle vertex counts quite differently, but both outperform FPGA execution by a large factor, as in SSSP. Our absolute figures for FPGA execution show time per iteration for SO^T of around 4 times the equivalent for LJ, meaning sparsity/vertex count has a large effect as well.

Figure 6.4: Execution Time for U250 PR – CPU (Zheng/Intel) over FPGA, per Dataset and Framework.

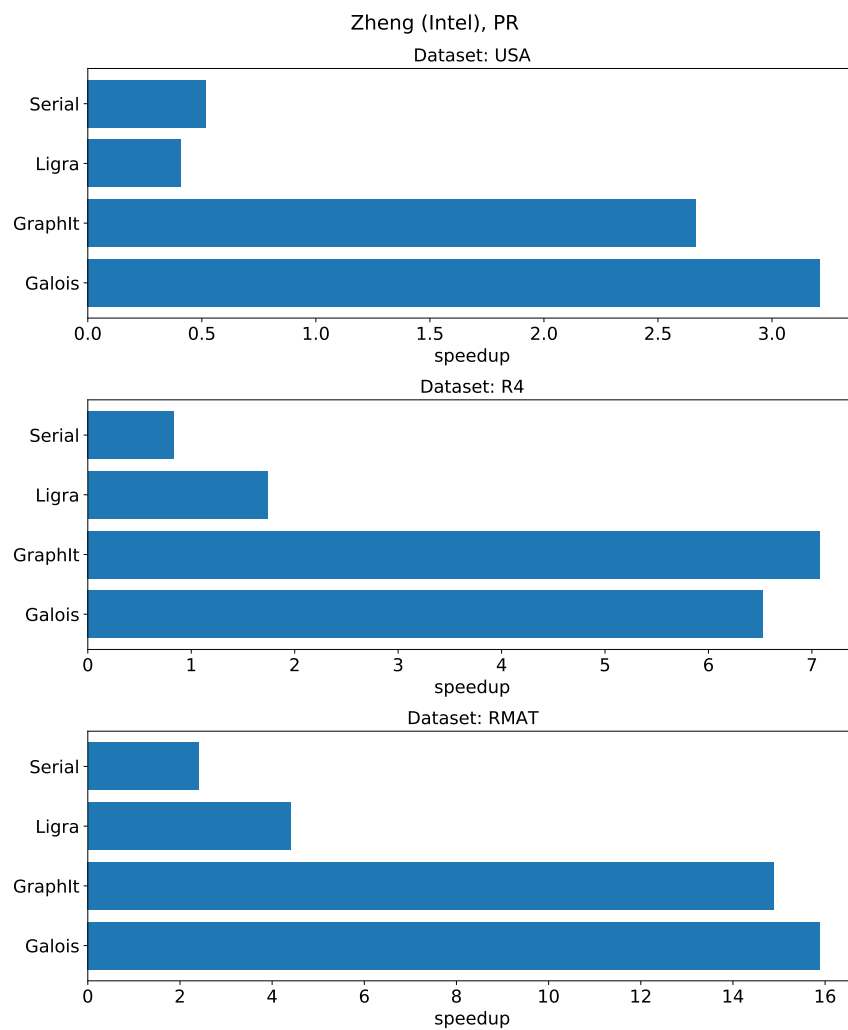
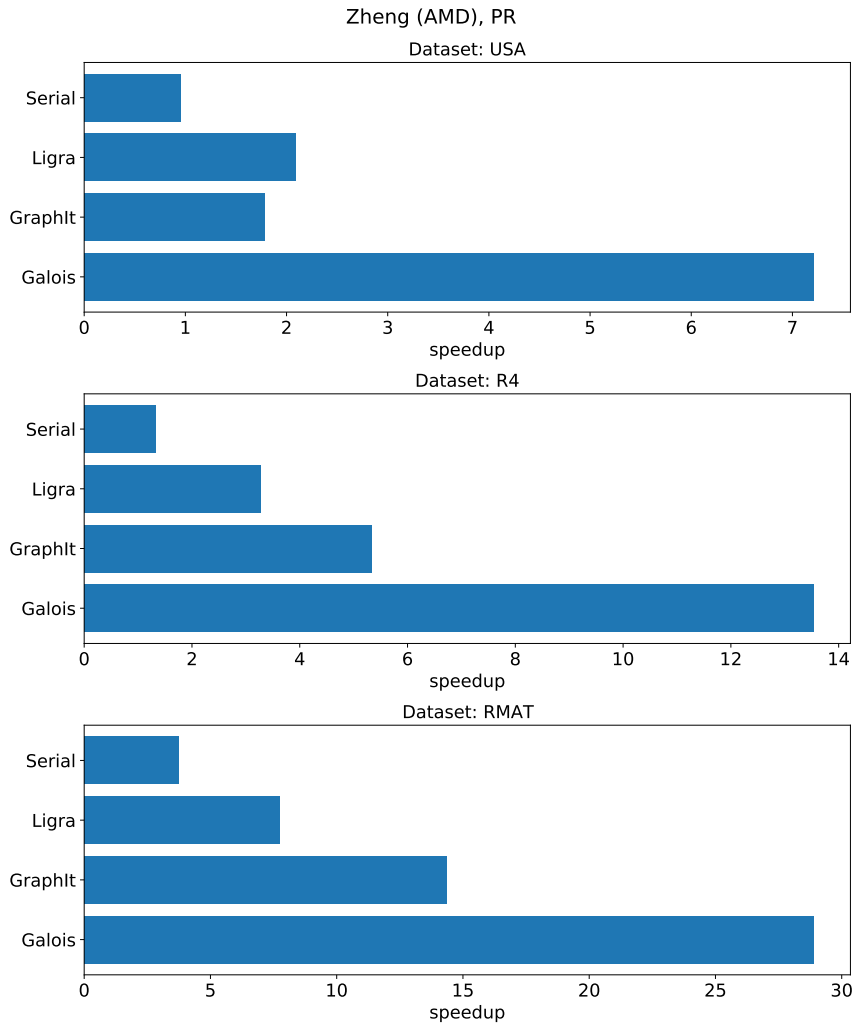


Figure 6.5: Execution Time for U250 PR – CPU (Zheng/AMD) over FPGA, per Dataset and Framework.



6.3.2.2 Alveo U280, 2 Channels

The PR-HBM2C implementation is the best-performant overall. However, the low number of banks assigned to the dataset ports make it so the SO graph (which is the largest in our PR tests) cannot be loaded onto the FPGA, so it is excluded.

The panorama in Figures 6.7, 6.6 and 6.8 is somewhat similar to that for U250 execution for USA, but is significantly better for the two denser matrices. At this point, an energy analysis could reveal this implementation outranks some CPU frameworks, energy-wise.

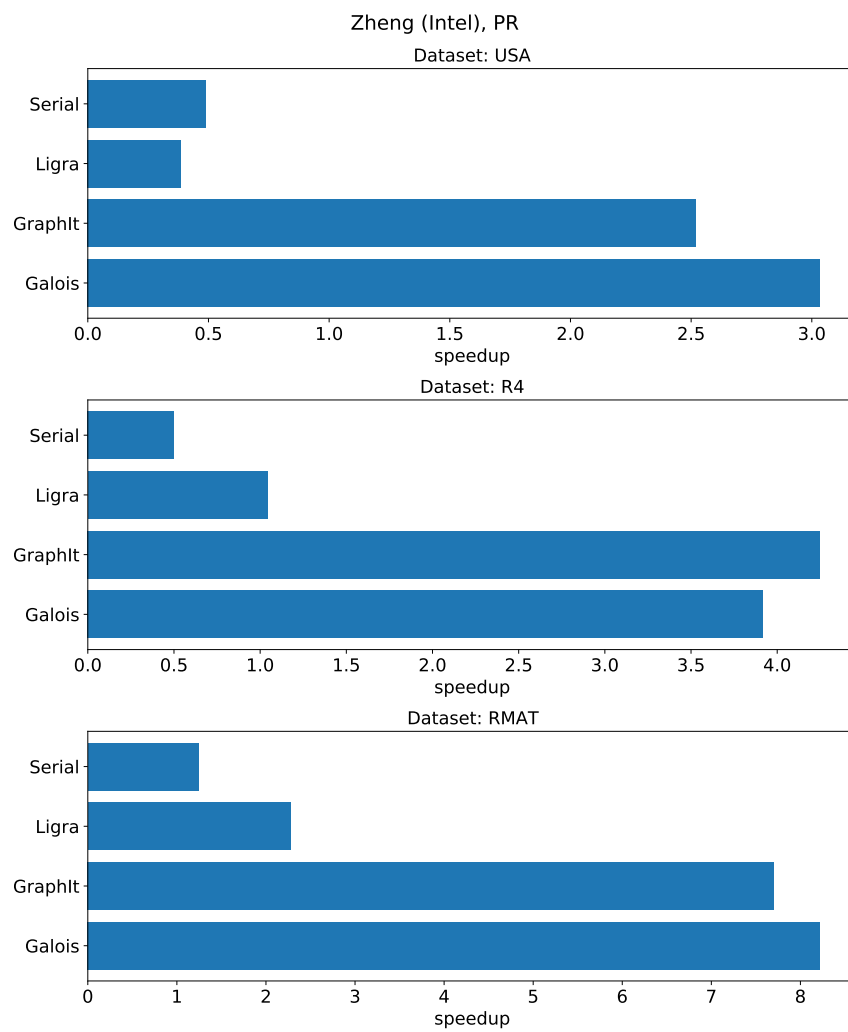


Figure 6.6: Execution Time for U280 PR, 2 Channels – CPU (Zheng/Intel) over FPGA, per Dataset and Framework

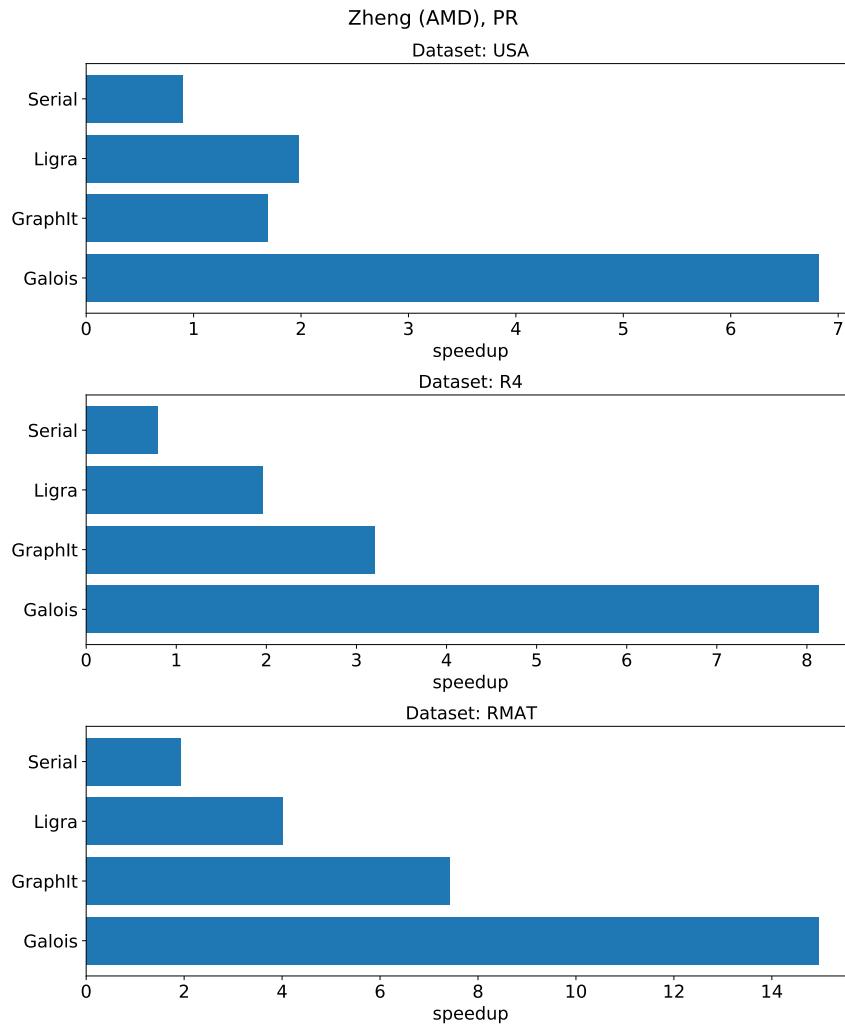


Figure 6.7: Execution Time for U280 PR, 2 Channels – CPU (Zheng/AMD) over FPGA, per Dataset and Framework

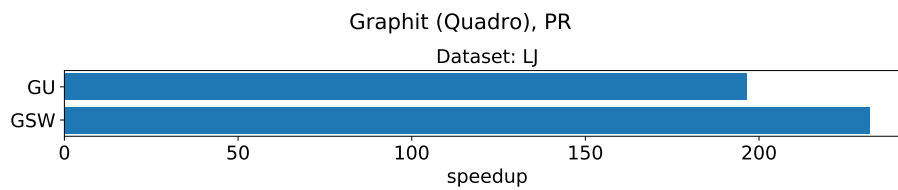


Figure 6.8: Execution time for U280 PR, 2 Channels – GPU (Graphit/Quadro) over FPGA, per Dataset and Framework.

6.3.2.3 Alveo U280, 6 Channels

Figure 6.9 shows PR-HBM6C outranks Alveo U250 execution overall, except for USA. This may indicate a density or a vertex/edge count issue. The poor performance versus GPU shown in Figure 6.10 may point to the latter.

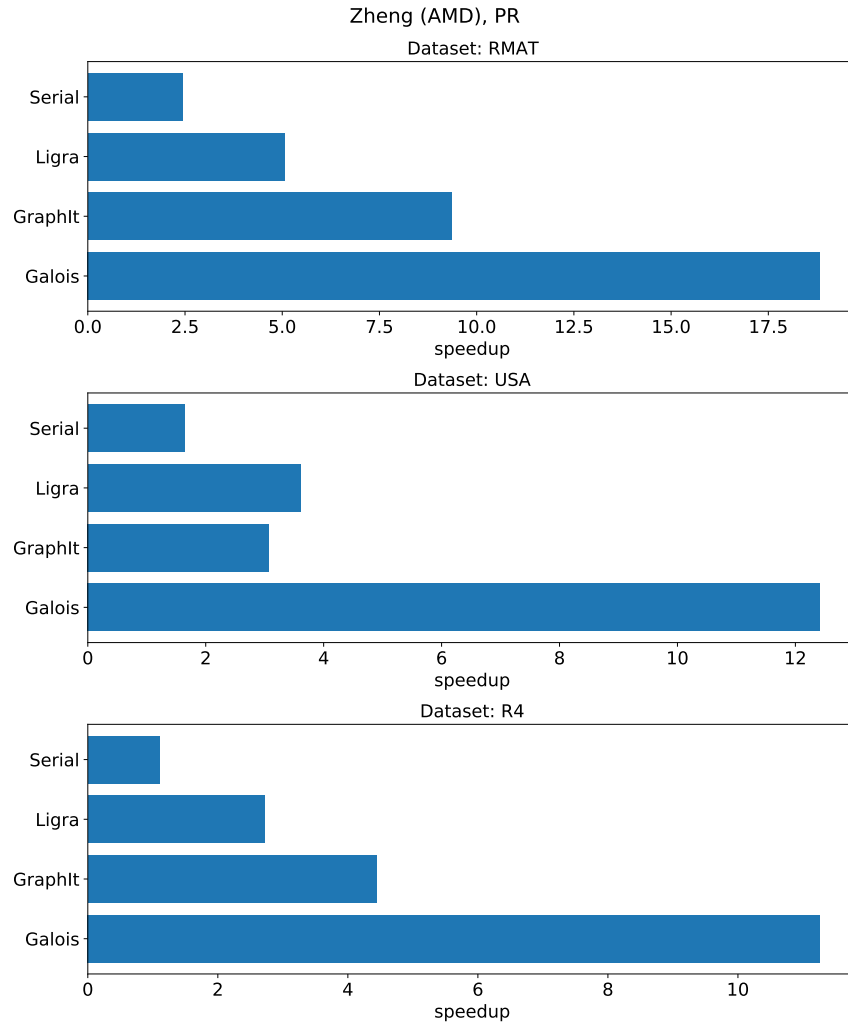


Figure 6.9: Execution Time for U280 PR, 6 Channels – CPU (Zheng/AMD) over FPGA, per Dataset and Framework

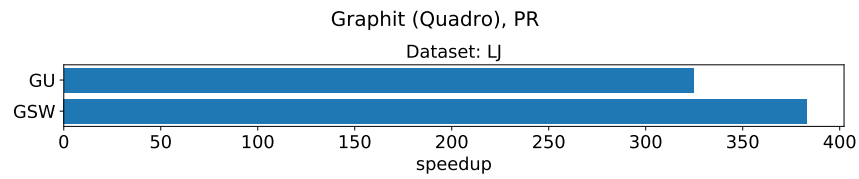


Figure 6.10: Execution time for U280 PR, 6 Channels – GPU over FPGA, per Dataset and Framework.

6.3.2.4 Alveo U280, Jumbo

Our PR-HBMJ implementation is able to load the SO graph, so we again show results versus GPU in this instance, as seen in 6.11. All results versus GPU outrank PR-DDR, but results for GPU/LJ are worse than in PR-HBMC2. Results versus CPU are similarly worse as seen in Figures 6.13 and 6.12.

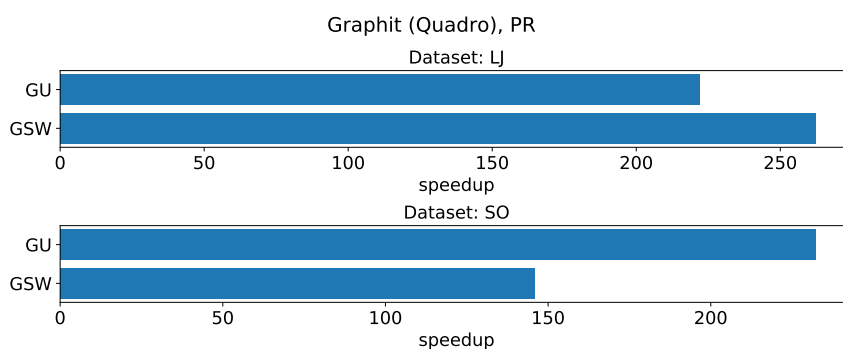


Figure 6.11: Execution Time for U280 PR, Jumbo – GPU over FPGA, per Dataset and Framework

This suboptimal performance may be caused by HBM port-to-bank mapping particularities; we will discuss these again in Subsection [6.3.3.2](#).

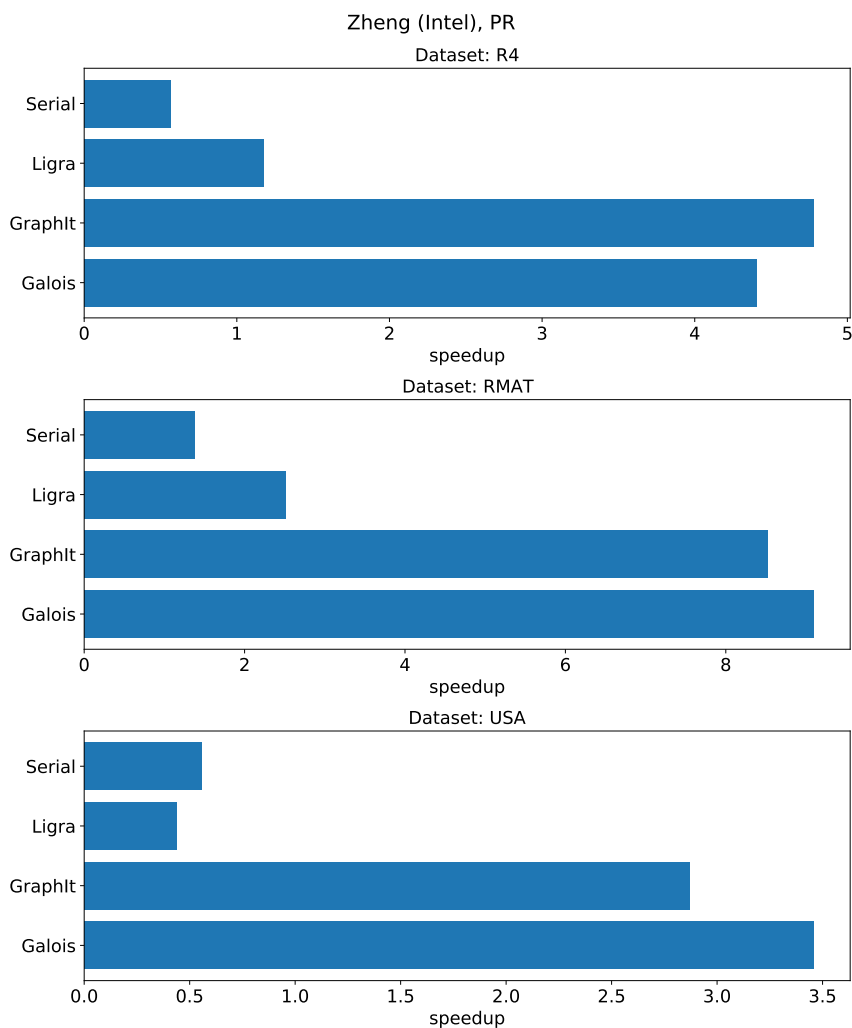


Figure 6.12: Execution Time for U280 PR, Jumbo – CPU (Zheng/Intel) over FPGA, per Dataset and Framework

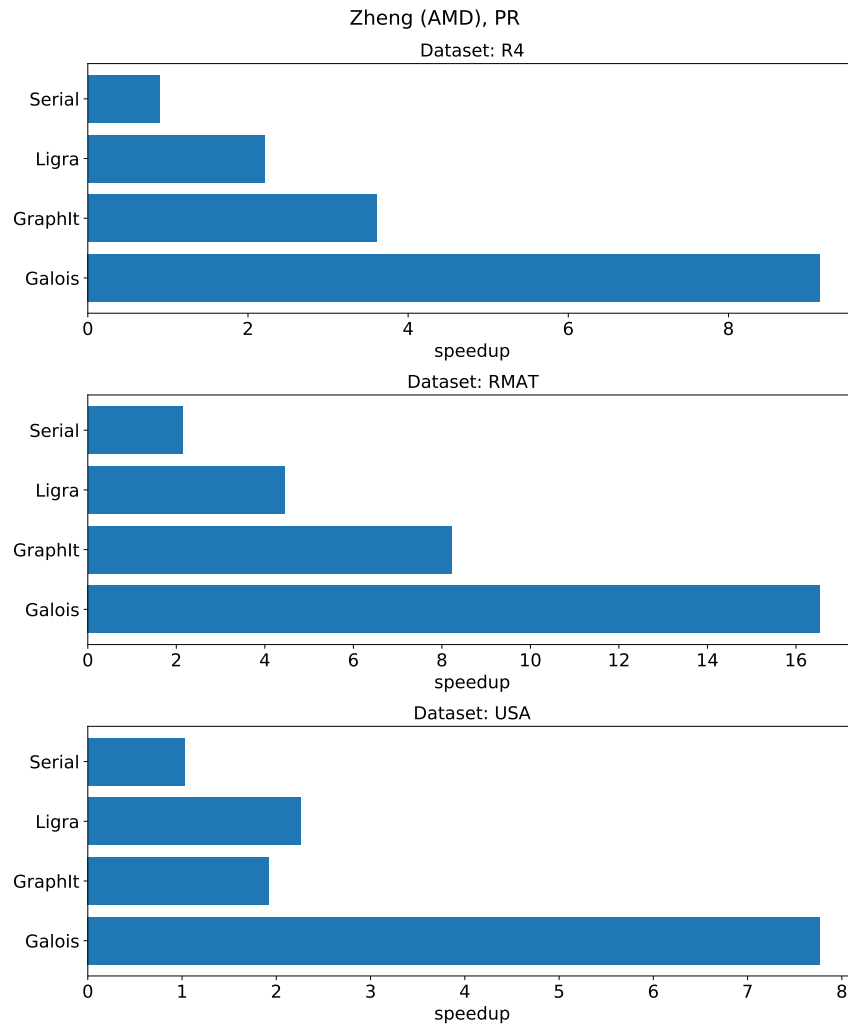


Figure 6.13: Execution Time for U280 PR, Jumbo – CPU (Zheng/AMD) over FPGA, per Dataset and Framework

6.3.3 Distinctiveness Centrality 1

The following results show the progression in optimisations from our DC1 baseline to our “flagship” variants by means of displaying the speedup in kernel time between consecutive optimisations.

Note that the P optimisation never appears by itself, but always in conjunction with the D optimisation. We found that the P optimisation produced incorrect results in the HBM8OD variant, likely related to the usage of all 32 HBM banks, which is stated to have a chance of causing memory corruption (recall Subsection 4.2.3). Adding the D optimisation fixes this (perhaps due to a decrease in area consumption or in memory access contention), so we have scrapped all *OD results.

6.3.3.1 Alveo U250 Progression

We will now evaluate the progression in performance between the DDR/Alveo U250 variants. Figure 6.14 shows the first optimisation: increasing the number of CUs to four in order to exploit data parallelism. As would be expected, the maximum achieved speedup is close to 4. However, in graphs with high edge-to-vertex ratios, this is reduced. This is very likely due to the inherent unevenness of the edge partitioning scheme mentioned in Subsection 5.4.5.2, indicating larger degree variance in these graphs which causes sub-optimal performance.

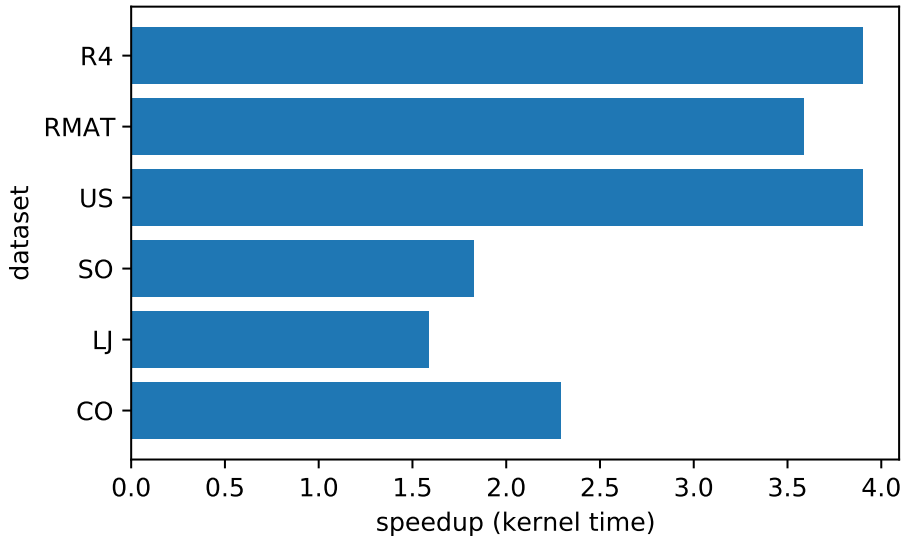


Figure 6.14: Kernel Time for DC1 – DDR4 over DDR1, per Dataset

Figure 6.15 shows the results of the application of the D and P optimisations. A note on the behaviour of the baseline, DDR4, HBM1J7, and HBM8: if we assume the existence of some maximum degree constraint, then these variants may be modelled as $O(V * k) = O(V)$. However, if we *do not* assume the existence of this constraint, then the maximum degree of a vertex is given by $V - 1$, so the variants are modelled as $O(V(V - 1)) = O(V^2)$. On the other hand, all D variants are unambiguously modelled as $O(E)$. It follows that speedup *should* be linearly correlated to some function in $V^k/E, k \in [1, 2]$, where k in some way increases as a function of the sparsity of the graph.

From another angle, we may model the performance of the baseline as $\delta_1 V + E$, and of the D optimised kernels as $\delta_2 + E$, where the δ variables signify the latency of the outermost loop. Ignoring δ_2 as it is only incurred once, this gives us a more manageable formula of $\delta_1 V/E + 1$, which is virtually inversely proportional to E/V . Table 6.2 thus gives the expected ascending order of: CO, SO^T, LJ, RMAT, R4, and US.

This ordering is broken in Figure 6.15: LJ shows a higher speedup than RMAT. We believe this is, again, caused by unbalanced partitioning. Switching to an edge-oriented method, along with benefiting graphs with high vertex counts, also benefits graphs with irregular edge distributions: in

this instance, the latter effect overwhelms the former for LJ. Notice that Figure 6.16, which shows DDR4ODP *vs.* DDR1 speedup, shows no such effect, as DDR1 does not partition the dataset.

An interesting factor is that this performance increase comes at a relatively low resource usage cost: resource usage often increases sub-proportionally to the number of CUs, as shown in Appendix A.

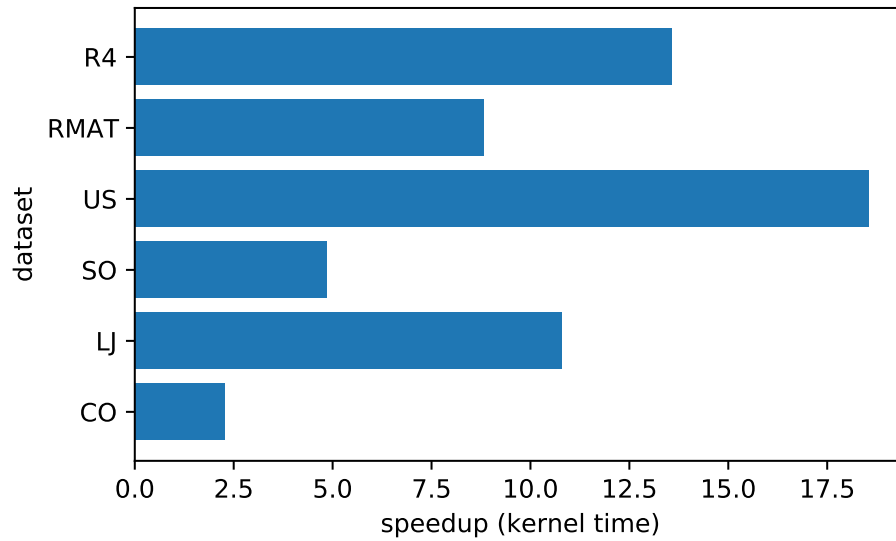


Figure 6.15: Kernel Time for DC1 – DDR4ODP over DDR4, per Dataset

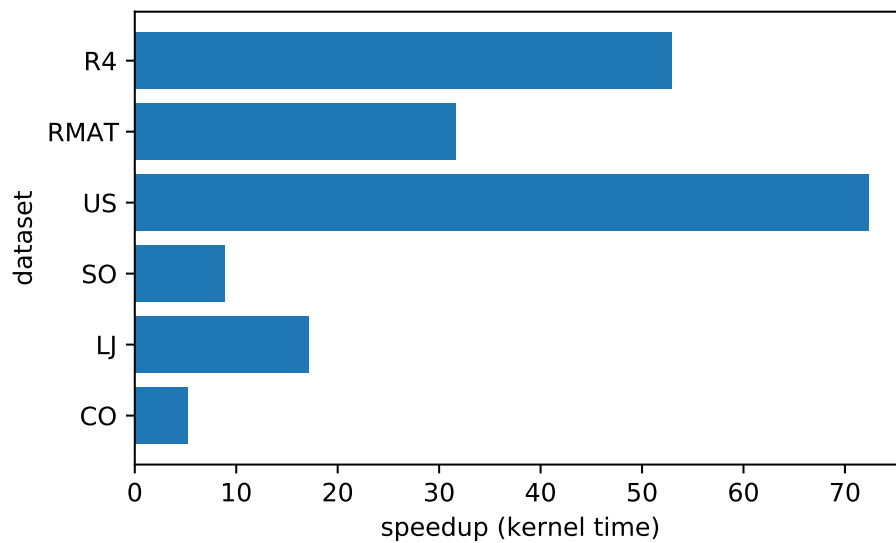


Figure 6.16: Kernel Time for DC1 – DDR4ODP over DDR1, per Dataset

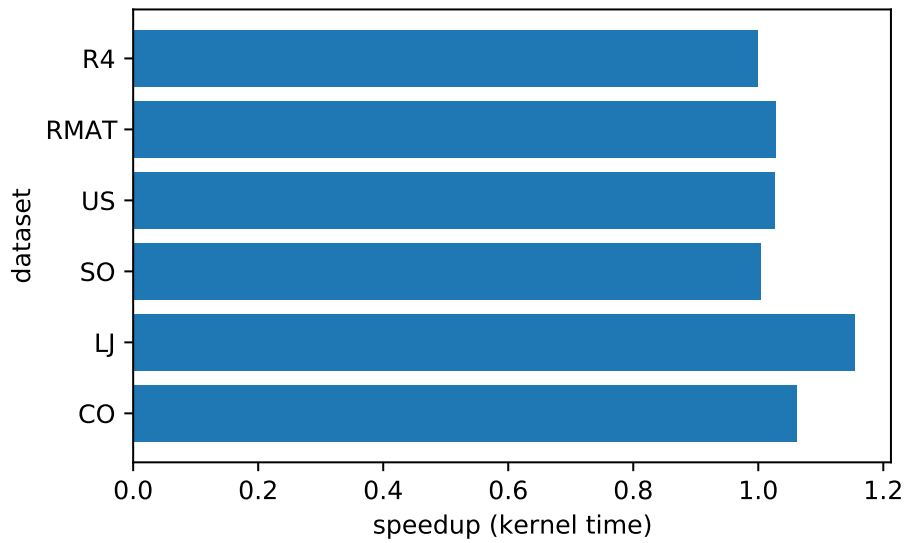


Figure 6.17: Kernel Time for DC1 – DDR4ODPCB1 over DDR4ODP, per Dataset

Figure 6.17 shows our first results for the CB optimisation. As expected, kernel execution times are not greatly affected: the CB optimisation mainly consists of interleaving the accumulate phase with the device execution phase, so benefits are only really visible when assessing total execution time (see Appendix B). Figure 6.18 returns to significant speedups: the unrolled version is able to process a chunk faster, offsetting even the cost of generating and writing the degree queue. Execution of dataset US is sped up significantly, likely because edges of the same vertex are often in spatial proximity, making degree generation faster.

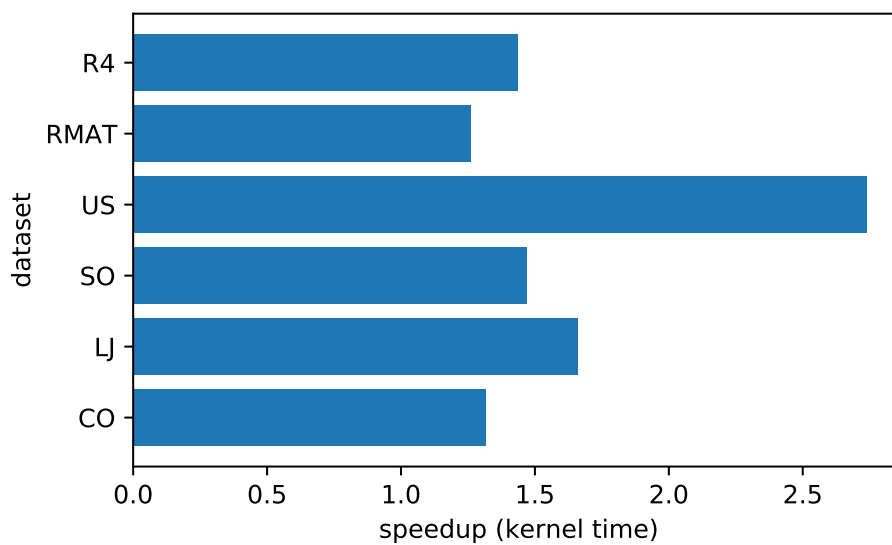


Figure 6.18: Kernel Time for DC1 – DDR4ODPUCB1 over DDR4ODPCB1, per Dataset

6.3.3.2 Alveo U280 Progression

Figure 6.19 shows the performance of our first HBM-enabled variant, HBM1J7, against the baseline. Ideally, since we are execute-bound to one output per cycle, we would expect the speedup to roughly equal 1 in all instances. However, some datasets show a small performance decrease. We have not explored this effect in detail, but we suspect it is related to one of the many particularities of HBM: several ports span over 4 banks (7) while not being aligned to 4 (bank 0, bank 4, ...). This may cause contention issues in the switch interconnect.

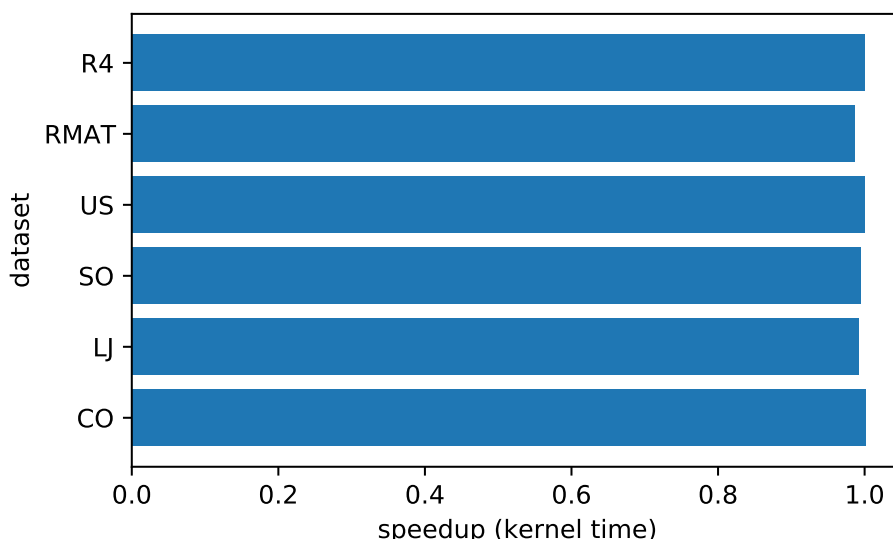


Figure 6.19: Kernel Time for DC1 – HBM1J7 over DDR1, per Dataset

Figures 6.20 and 6.21 paint a similar picture to DDR4ODP *vs.* DDR4 *vs.* DDR1: the theoretical maximum is almost achieved for some datasets, while others are greatly affected by the partitioning issue. We would expect the former figure to show approximately 4x the speedup of DDR4 *vs.* DDR1, and the latter figure the same speedup as DDR4ODP *vs.* DDR4. Both underperform in this regard, however. The issue may be yet another particularity of HBM: as each port has less banks mapped to it, the number of consecutive accesses to the same bank group is higher, which may incur an overhead (recall 4.2.3). Additionally, in the latter case, we may be witnessing diminishing returns of the D optimisation as partition sizes (and thus partition degree variance) grow smaller.

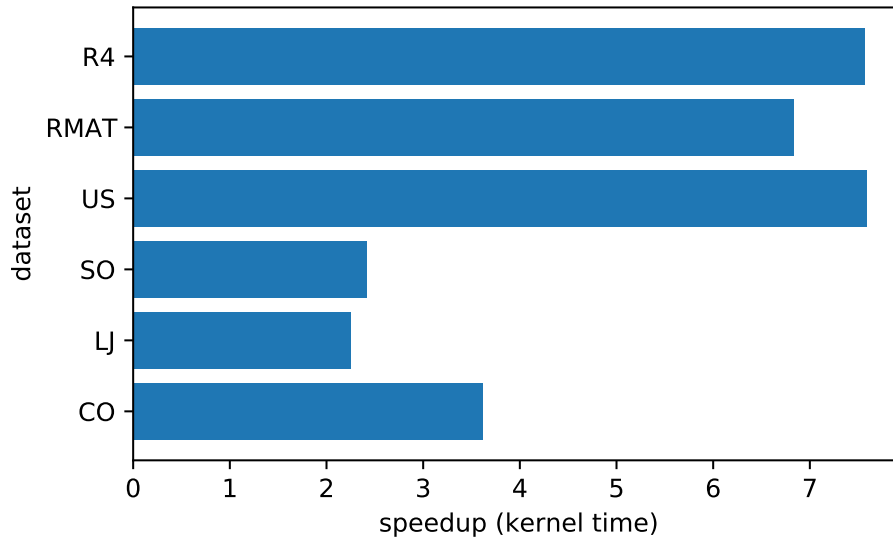


Figure 6.20: Kernel Time for DC1 – HBM8 over HBM1J7, per Dataset

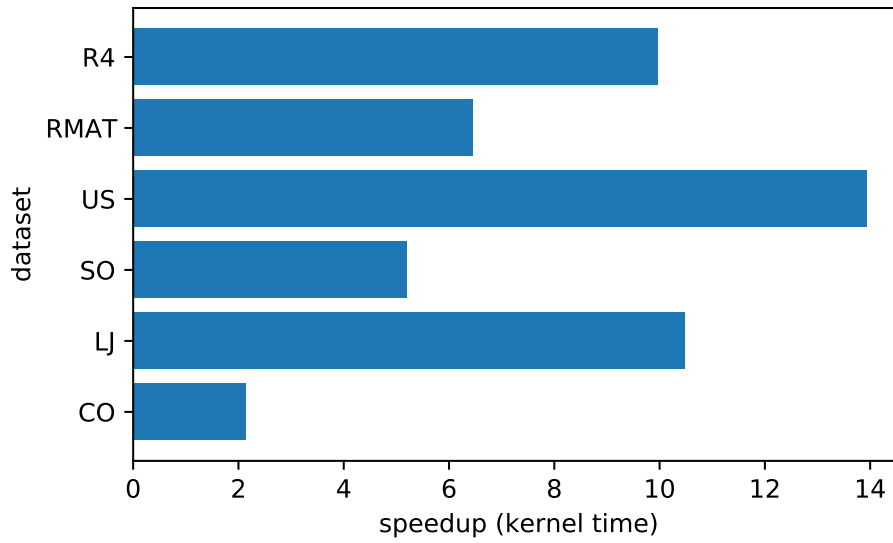


Figure 6.21: Kernel Time for DC1 – HBM8ODP over HBM8, per Dataset

On the other hand, our HBM10ODP variant shows speedups beyond the expected 1.25, as shown in Figure 6.22. Recall that HBM10ODP, out of necessity (in order to fit into the available 32 banks) removes the “vestigial” C queue. We suspect this may cause the compiler to better optimise reads to O .

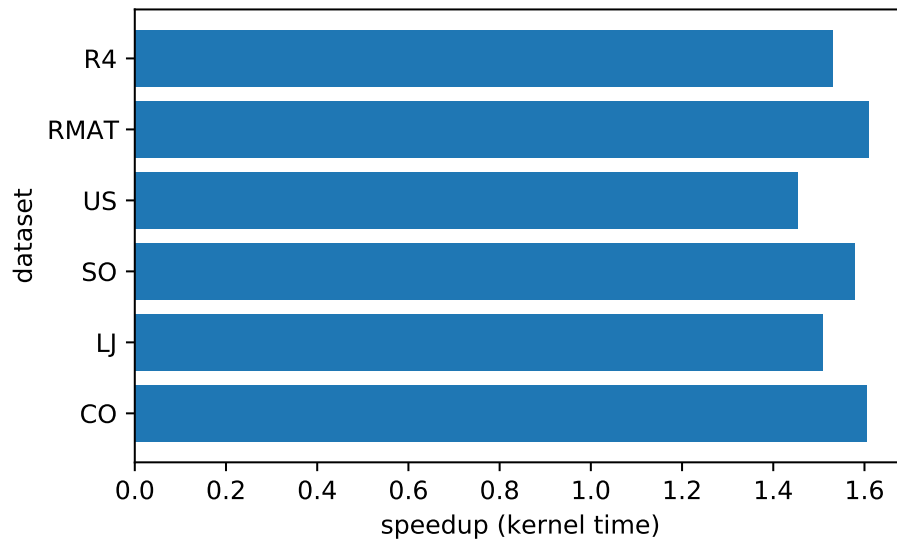


Figure 6.22: Kernel Time for DC1 – HBM10ODP over HBM8ODP, per Dataset

The result of introducing our CB optimisation, shown in Figure 6.23, is similar to that for its DDR4 equivalent. In contrast, the introduction of the U optimisation shown in Figure 6.24 results in significant *slowdowns* for all datasets except US. This is likely due to a parallelism limit: our program is now attempting to process 10 chunks in parallel, *including degree queue generation*, and may not have sufficient resources to do so, slowing down kernel enqueues. Nonetheless, the panorama is similar to that for DDR4: total execution times are lower than any other variant except HBM10ODPCB1, and lower *overall* compared to the latter (see Appendix B.3).

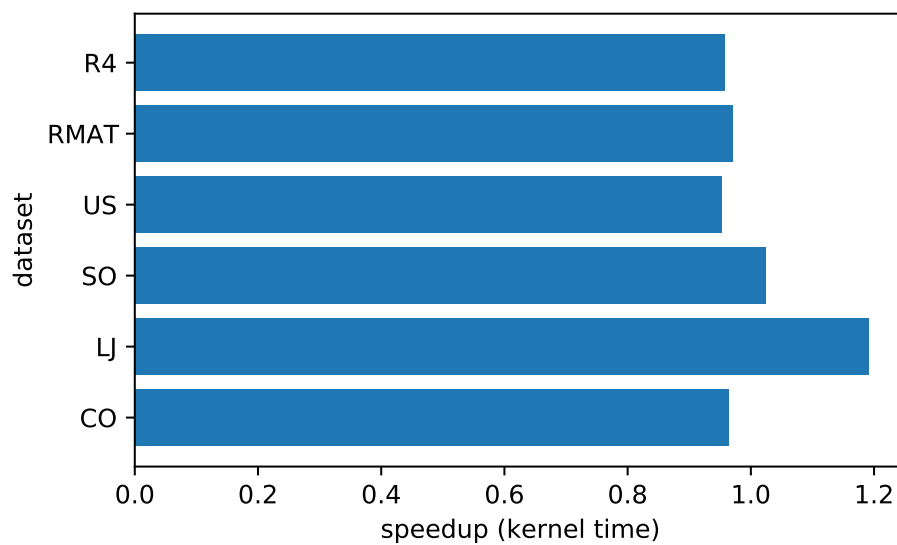


Figure 6.23: Kernel Time for DC1 – HBM10ODPCB1 over HBM10ODP, per Dataset

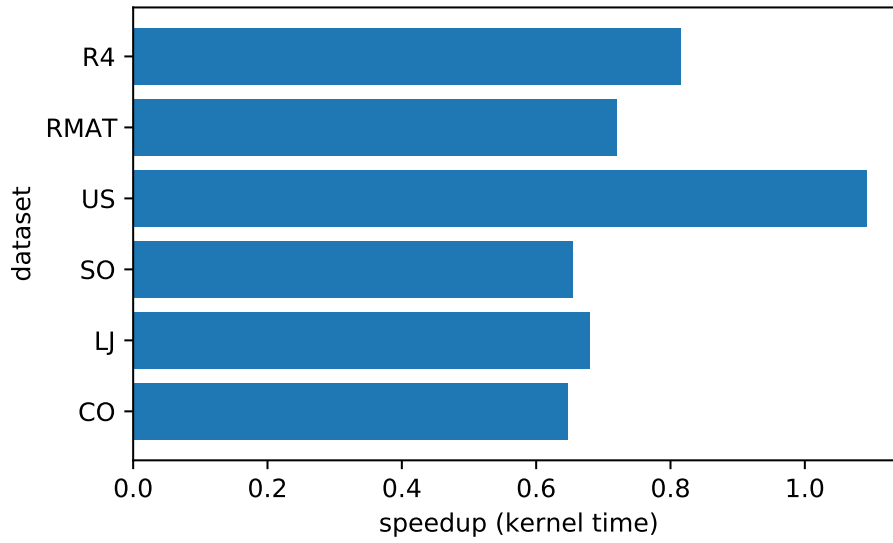


Figure 6.24: Kernel Time for DC1 – HBM10ODPUCB1 over HBM10ODPCB1, per Dataset

6.3.3.3 ODPUCB1 Variants, Untimed Degree Queue Generation

We analyse the impact of removing the degree generation phase from the timed enqueue loop section in Figures 6.25 and 6.26. Note that doing so, in principle, should allow chunks to be enqueued faster, thus decreasing total kernel time – this analysis is meant to indicate the possible impact of moving the degree generation to the device (we are still sped down, however, by the effect of transferring an E sized queue).

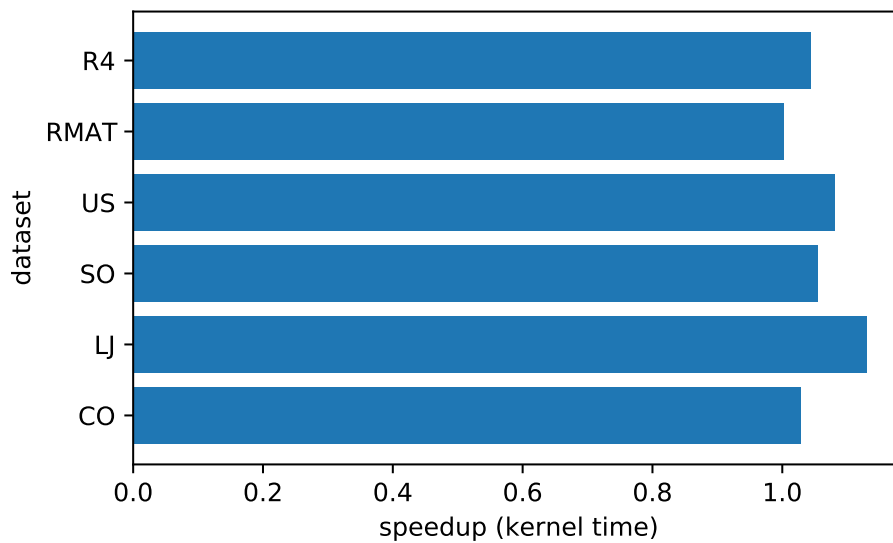


Figure 6.25: Kernel Time for DC1 – DDR4ODPUCB1 Excluding Degree Generation on the Enqueue Loop, over the Regular Variant, per Dataset

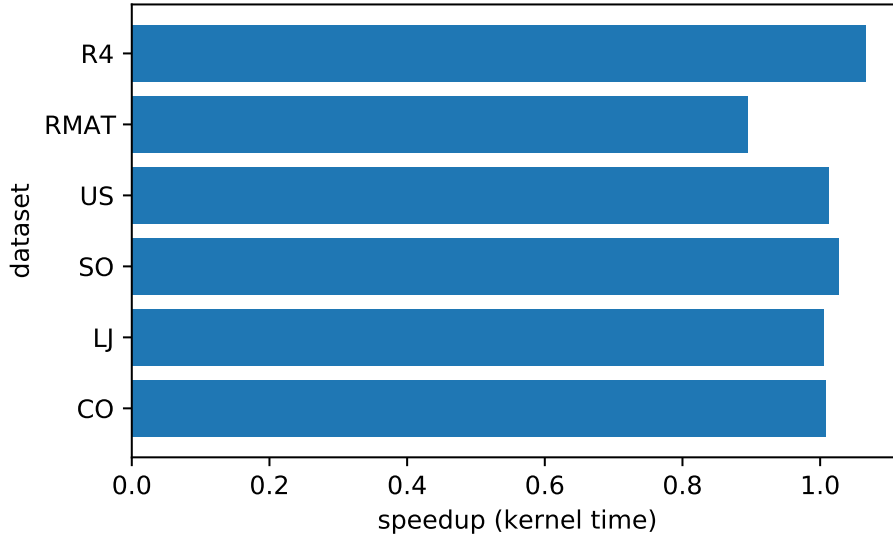


Figure 6.26: Kernel Time for DC1 – HBM10ODPUCB1 Excluding Degree Generation on the Enqueue Loop, over the Regular Variant, per Dataset

Overall, we see speedups surpassing 1, with one exception: the RMAT dataset leads to a slowdown, versus our regular ODPUCB1 variants. We are unable to explain this phenomenon, but suspect some property of RMAT (such as its relatively high edge-to-vertex ratio against its *total* edge count) may cause a staggering of chunks belonging to the same CU, this leading to less time spent waiting on the enqueue lock. Nonetheless, overall results suggest implementing the degree generation phase on the FPGA would have a positive impact on execution times.

6.3.3.4 Total Execution Time and Energy Expenditure Vs. C++ Implementation

We now enlarge our focus to the execution of the entire application, versus our control C++ implementation. Figures 6.27 and 6.28 show DC1 time- and energy-performance given as a relative speedup (or equivalent “energy factor”) of FPGA execution over CPU execution, for variants DDR1 and DDR4ODP respectively. As the FPGA implementation involves computation by the host, we include host execution times (as well as memory transfer times) and host execution power draw in our formula for total energy consumption. We are able to discriminate all stages during the simple sequential execution of DDR1 (except in the case of energy factor, where we measured power for device execution and accumulate execution), but the concurrent nature of DDR4ODP means we are only able to extract total *device time/energy* (the time span from the first write enqueue to the last read end).

One caveat is that power draw during memory transfer is larger, although the transfer itself is often very quick, especially for small datasets such as RMAT and R4, making power draw measurement during this stage difficult. Our total energy consumption numbers take into account this spike, but energy factor figures for RMAT and R4 may be skewed in favour of the CPU implementations. Similarly, FPGA execution times for all datasets except USA (11.3 s) are under

5 s. This is not sufficient to stabilise FPGA power draw after memory transfer, again possibly skewing results towards CPU execution in these instances.

Of note is our evaluation of the largest dataset, CO. Results for this dataset using DDR1 show the same preference for graphs with low vertex to edge ratios as explained in our D optimisation results. USA, by far the matrix with the highest ratio, has the worst speedup but the best energy factor, as power draw has had time to stabilise enough after memory transfer to overcome poor performance.

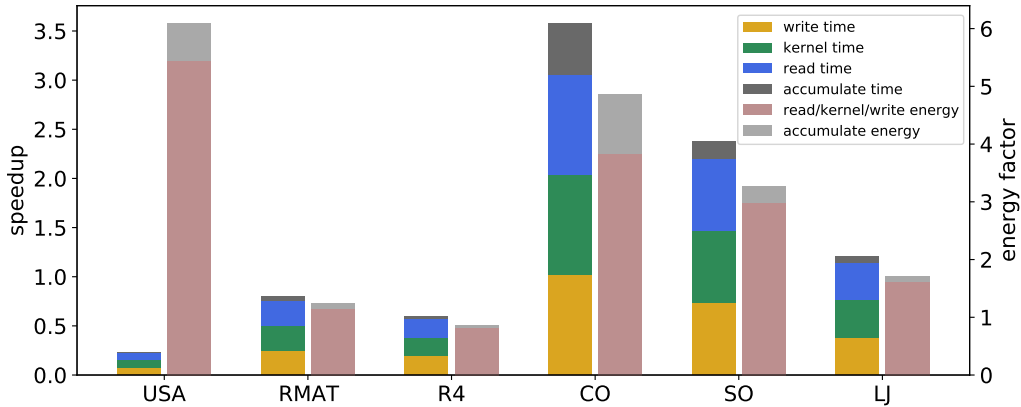


Figure 6.27: Total Execution Time and Total Energy Expenditure, Left to Right, for DC1 DDR1 – FPGA over CPU, per Dataset

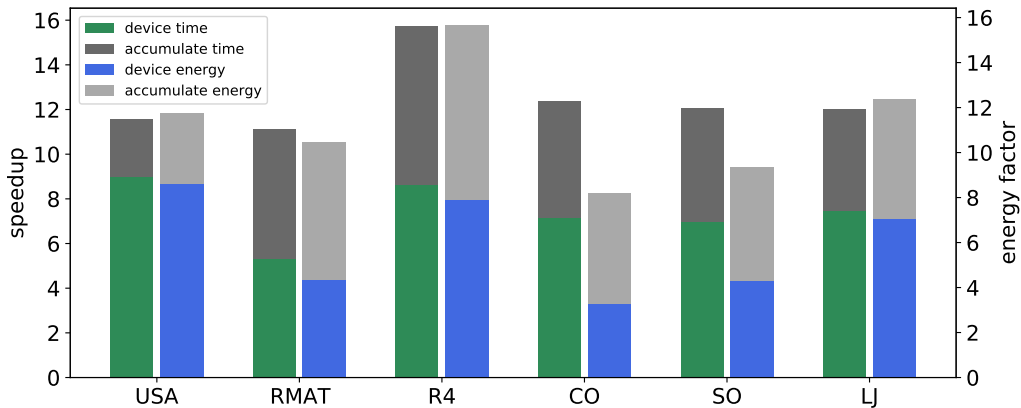


Figure 6.28: Total Execution Time and Total Energy Expenditure, Left to Right, for DC1 DDR4ODP – FPGA over CPU, per Dataset

DDR4ODP results give a semi-consistent approximate 11x-12.5x speedup, except for the R4 graph. One possible cause is memory locality: given a page size of 4 KiB, or 128 elements, we may plot the cumulative number of unique page requests for the R4, RMAT, and US datasets as shown in Figure 6.31. Also shown is the maximum page limit for the machine’s available RAM. This is not a dynamic analysis, and so does not show runtime behaviour, but we may note that R4 quickly requests approximately double the number of pages as RMAT, and goes past the page limit. If this behaviour is continuous, page faults may be abundant for R4. US, in comparison,

requests pages at a more measured pace, in blocks, suggesting old pages are no longer accessed much.

Figures 6.29 and 6.30 give results for our top variants. In this instance, as accumulation is interleaved with device execution, we become unable to discriminate this stage. Additionally, we do not have power data for any HBM variant, so we refrain from giving the energy factor in this instance.

DDR4ODPUCB1 outperforms HBM10ODPUCB1, which may come as a surprise, as the latter is capable of executing more chunks in parallel. However, as we’ve mentioned in Subsection 6.3.3.2, this is likely a parallelism issue – executing HBM10ODPUCB1 on our local machine would perhaps lead to better results. Note additionally that unrolling in this manner results in significantly higher resource usage, as shown in A.6.

Regardless, we are able to obtain significant speedups and energy factors for all datasets versus our control (recall, however, that these variants are compiled with a more recent version of GCC, which may have a small impact). Speedups are mostly directly proportional to those for DDR4ODP, with the exception of dataset USA, likely again related to data locality interacting with our optimised accumulate cycle.

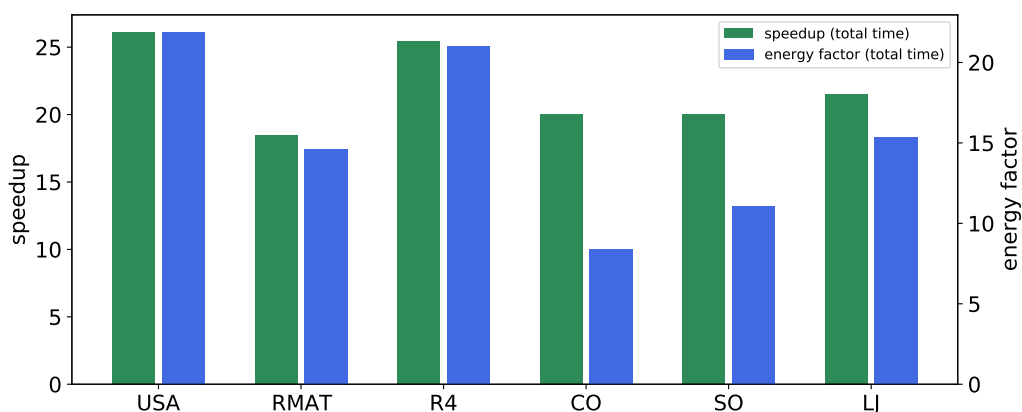


Figure 6.29: Total Execution Time and Total Energy Expenditure, Left to Right, for DC1 DDR4ODPUCB1 – FPGA over CPU, per Dataset

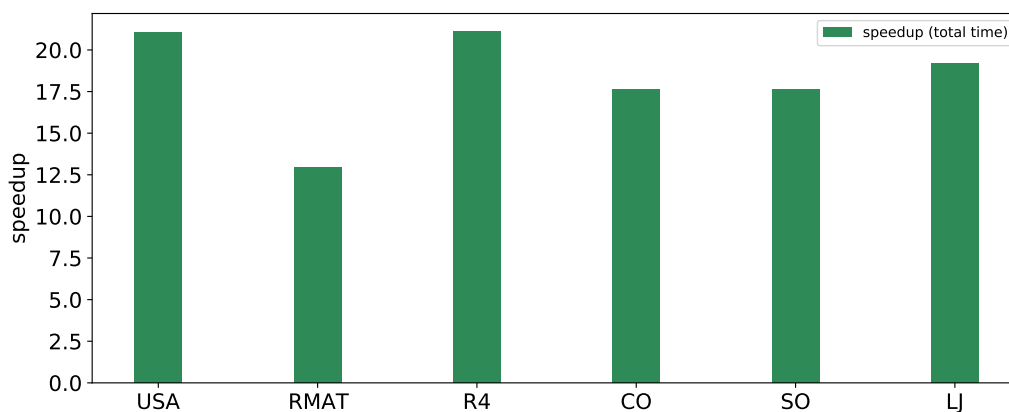


Figure 6.30: Total Execution Time for DC1 HBM10ODPUCB1 – FPGA over CPU, per Dataset

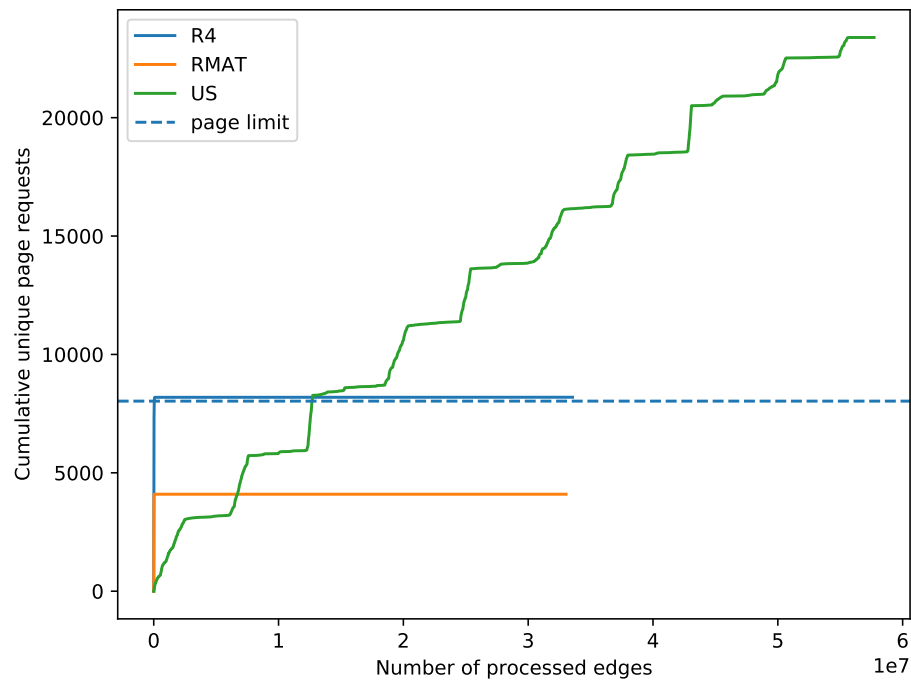


Figure 6.31: Cumulative Unique Page Requests For Writes *vs.* Number of Processed Edges for Datasets R4, RMAT, and US, for CPU DC1 Implementation

Chapter 7

Discussion and Conclusion

7.1 A Systematisation of HLS Abstractions and Optimisations

We propose a system of classification for HLS programming abstraction levels, based both on the overall impact of optimisations grouped within those abstraction levels, and on perceived difficulty from a software engineer’s perspective. Table 7.1 presents the system and gives examples of optimisations falling under each category.

We posit that knowledge acquisition rate is constant up to the Intermediate-2 stage (as prior software engineering knowledge can be leveraged), but increases per stage – notably, with a significant spike in Intermediate-2 (as *some* prior knowledge cannot be leveraged any longer). Later stages require new acquisition of hardware engineering knowledge, so effort required grows at a linear pace (such that Advanced-2 requires significantly more effort). Plugging in reasonable values for effort rate yields Figure 7.1, which plots a possible cumulative developer effort to achieve the knowledge required to effectively use optimisations in each category.

Based on this concept, we propose that vendors of HLS tools targeted to software engineers/domain experts set an absolute limit beyond which the user is *not* expected to have to proceed. Our own experience leads us to believe this limit lies somewhere between Intermediate-2 and Advanced-1. We note our own DC1 optimisations mostly lie in Intermediate-1, with occasional ventures into the next category. There are certainly more optimisations possible (we note some of these in Section 7.4), but application optimisation rapidly becomes intractable if the effort to learn and implement such an optimisation brings some development value function to zero. Further optimisations should be *leveraged* by the domain expert, but implemented by FPGA/HLS experts – thus bringing the optimisation to a lower category, and back to the set of optimisations reachable by the user. This may be done via libraries, frameworks, automatic Design Space Exploration via static or dynamic analysis, etc.

A possible strategy for HLS vendors is thus to categorise optimisations found in a wide array of domain-specific accelerated application code (either using some sort of system like our own or another if more adequate), and use the above strategies to systematically lower the category of frequently encountered optimisation patterns.

Table 7.1: Classification System for HLS Programming Abstraction Levels

Name	Description	Examples
Software Paradigm	Direct usage of CPU-specific code, with little or no additional configuration or compiler directives/pragmas	N/A
Beginner	Simple optimisations, with little to no comprehension effort required, such as those described in [77]	Grouping AXI interface ports into separate bundles for concurrency, pipelining inner execution loops, usage of dataflow when trivial
Intermediate-1	Abstractions which require considering the underlying hardware generation process but have close parallels to traditional CPU programming	Pipelining non-embarrassingly parallel loops and functions, partial and full unrolling, kernel replication via multiple CUs, usage of dataflow when non-trivial
Intermediate-2	Abstractions which require considering the underlying hardware generation process, but do <i>not</i> have close parallels to CPU programming	Optimising memory bank mapping and usage (esp. for HBM), data structure/complex refactoring in order to support pipelining and unrolling, memory access refactoring in order to support full unrolling (this includes coaxing the compiler to detect burst accesses)
Advanced-1	Abstraction level lowered towards the programmable fabric and usage of discrete logic blocks	Simple explicit usage of local memory such as URAM and BRAM, code specifically targeting embedded DSPs
Advanced-2	Usage of common hardware development abstractions and concepts	Heavy usage of local memory for optimisation, explicit programming to generate and use structures such as ping-pong buffers or shift registers, integration with and configuration of RTL IP
Hardware Paradigm	HLS/RTL mixed coding with heavy reliance on hardware-engineering concepts applied to the HLS model	Implementing complex discrete components or IP using HLS, writing, configuring, and implementing RTL IP

Note that FPGA-unfriendliness makes optimisation difficult, so handling FPGA-unfriendly problems necessarily pushes the developer to lower/less abstract categories.

7.2 DC1-CSR Properties

We’ve previously mentioned, in Section 1.4, a set of interesting properties our DC1-CSR formulation possesses. We elaborate on these.

7.2.1 Data-Parallelism

All variants, including edge-orientation optimised variants, are fully data-parallel at the vertex level. Computation of the edge set of any slice of the vertex set can be offloaded, in a completely

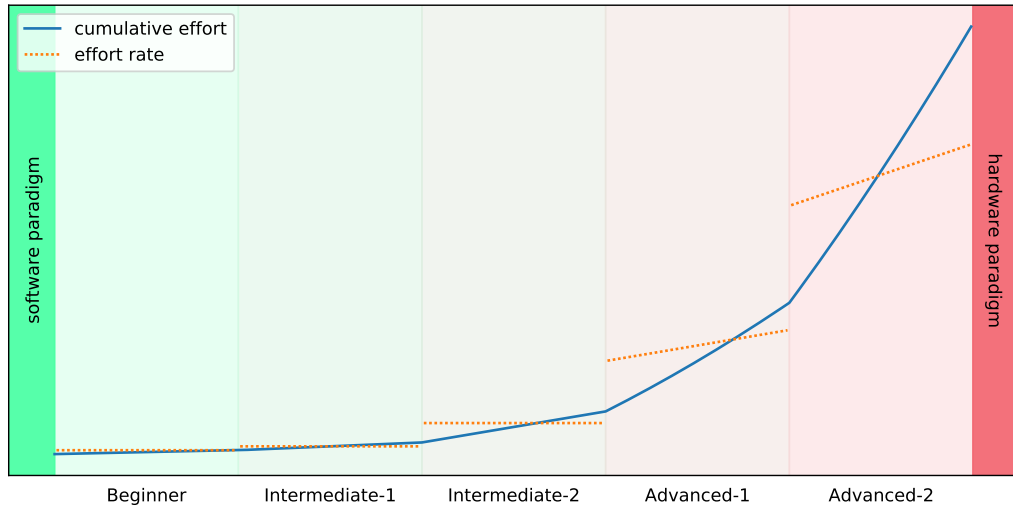


Figure 7.1: Possible Mapping of Abstraction Levels to Developer Knowledge Acquisition Effort Rate

independent fashion, to any compute unit. This gives the formulation good scaling potential.

7.2.2 Streamability

All variants are streaming algorithms. This makes analysis (both formal and otherwise), extension, and implementation simpler.

7.2.3 Dynamicism

A *dynamic graph* is a graph structure such that edges or vertices may be added or deleted at any time. Dynamic graph algorithms focus on efficiently updating the result of the algorithm after an update, rather than re-run the entire algorithm [20].

An algorithm is said to be incremental if it only allows insertions, or decremental if it only allows deletions. If both operations are supported, the algorithm is said to be fully dynamic [20].

DC1-CSR is immediately incremental for vertices: to add a vertex, simply send its data to the FPGA. Edge insertions are more difficult – since the vertex’s degree changes – and so are edge deletions. Vertex deletions are somewhat simpler, as we must only negate the vertex’s effect on the graph.

Nevertheless, a simple (naive) fully dynamic scheme may be formulated for DC1-CSR:

- Vertex deletions: for each edge weight in vertex i , obtain its symmetric. Create a new virtual vertex with this data and send it to the FPGA. Update, and either alter the result array to take into account the deleted vertex or keep track of the deleted vertex’s offset for later updates.
- Vertex insertions: Send new vertex data.
- Edge deletions: Delete the vertex, and re-send the altered vertex data.

- Edge insertions: Delete the vertex, and re-send the altered vertex data.
- Edge updates: Either delete and then re-insert the updated edge, or assemble new vertex data such that all edge weights are null, save for the updated edge's (which should be $w_{new} - w_{old}$), and send this vertex data (we may do this because the vertex's degree does not change).

Many of these alterations could be performed by the host only, via direct alteration of the result array. However, if the number of updates is large, updating via the FPGA is likely more efficient (which is what prompted the FPGA implementation in the first place).

Note that further mathematical refinement to this scheme is very likely possible: we simply wish to demonstrate that DC1-CSR is dynamic.

7.2.4 Possible Iterative Variants

Iterative graph algorithms are common. The purpose of iterating is to produce sequentially better values, often until convergence [107] (see, for instance, the iterative formulation of PageRank).

DC1-CSR lends itself to iteration: with the addition of a new E -sized queue for a new edge property which is based on its source vertex's value¹, we may backfeed results onto the FPGA, which then takes the current edge value into account when computing the new value. This allows updates to propagate throughout the graph, such that the local (neighbour-only) DC1 algorithm is made into a global algorithm.

The precise manner in which previous values should affect new values is largely the subject of graph theory, which is not the main focus of this work. Nonetheless, iterative variants *are* possible, and may possess interesting properties.

7.3 Conclusion

We have implemented FPGA-driven applications for SSSP, PageRank, and DC1. We reiterate that, to the best of our knowledge, implementation of SSSP and PR is truly only viable, from a software engineer/domain-expert's viewpoint, using high-level tools – and of these, the only we found acceptable were the Vitis Libraries components we use, although we still believe ThunderGP holds good potential.

Our evaluation of SSSP presents a clear negative to the question of competitiveness versus CPU/GPU frameworks. Execution times are at least one order of magnitude worse versus *any* framework, so even energy-efficiency may be lower.

Our evaluation of PR paints a somewhat brighter picture. While results versus GPU are still drastically worse, execution time versus CPU is quite competitive, or even *better* in some rare instances. This leads us to believe FPGA PR acceleration currently is or may become viable in

¹Or by repurposing the V queue, or by adding a new N sized queue for vertex properties, but this requires random access.

the near future. From an energy-efficiency standpoint, some instances likely outperform CPU execution as well, but further study is required.

Regarding DC1: this is a novel algorithm, so application is still uncommon. We must stress the interesting properties mentioned in Section 7.2 which may increase usefulness even further whenever the metric is deemed acceptable in some domain – in this instance, we must therefore measure viability not only of the accelerator, but of generalising the metric itself to other domains; a DC1 implementation which gives worse results in the case of NavSys, for instance, may still be *better* if it produces those results faster than slower algorithms like BC, or provides easy dynamic updates thanks to its properties.

Our evaluation of DC1 was performed against a naive control: a single-thread C++ implementation. Nonetheless, scaling the control’s results by a factor of 10 still gives the advantage to the FPGA accelerator (in terms of time-performance). As no other method of accelerating DC1 is known to us, we may safely state that DC1 FPGA acceleration is viable.

Overall, we must conclude that FPGA-unfriendliness plays a large part in FPGA acceleration viability. This is by no means a formal analysis, but one may note SSSP requires large amounts of random accesses, as well as global knowledge of the graph, and performs quite poorly. PR requires less amounts of random accesses, is well suited to caching techniques, and performs acceptably, even yielding an advantage in some situations. DC1-CSR requires no random memory accesses due to its streaming model, so optimisation is simpler and results better – in fact, we would hesitate to classify DC1-CSR as an unfriendly problem.

There is thus a two-pronged conclusion: domain-experts must take FPGA-unfriendliness into account when considering the viability of an HLS FPGA implementation (in other words, the computation model must be a “good fit” to the FPGA model), but vendors and academia should also strive to design schemes which transform “bad fits” into “good fits”. One such scheme is presented in Section 7.1.

We do note that the time is right for research into FPGA graph acceleration. Usage of the technology is increasing, new advances in areas such as memory technology may come to popularise it even further, and there remains quite a lot to discover as regards to what FPGA acceleration can deliver for specific HPC workloads.

FPGAs themselves have a long history; in fact, they have paralleled the development of hardware engineering, including current hallmarks such as EDA. In recent years, however, FPGA technology has become accessible to another world – that of software developers – mostly thanks to the ultimate success of the HLS approach. FPGAs are now penetrating fields such as HPC as first-class citizens, much like GPUs did before.

The state-of-the-art in FPGA graph acceleration is also shifting: where single-issue, low-level, RTL implementations dominated, now appear high-level frameworks, and, most interestingly, *HLS-based* frameworks. This may indicate the validity of what we argue: RTL is reaching, or has reached, an “abstraction wall”. HLS is a key tool in overcoming this wall, but the abstraction level must be raised even further – and this means there is need of high-level frameworks, tools,

and libraries for specific application domains².

Discussion on what FPGAs are *suitable* for is commonplace, especially as the focus for HLS shifted from strict dataflow to general-purpose programming. As we have mentioned, we believe higher-level tooling is key if FPGAs are to ever be suited for unfriendly applications (and thus to general-purpose computing at large). Tooling *over* HLS (not involving a library) is, in essence, a shifting of the HLS source language from C/C++ to domain-specific DSLs. At another level, it has been proposed to shift the HLS *target* from Verilog/VHDL to an intermediate representation such as MLIR/Calyx [61]. This covers levels 2 and 3 of our unfriendliness classification (see Section 2.7). Improvements at the hardware level are also a possibility – HBM, for instance, has the potential to significantly benefit certain applications, and, as mentioned, future advancements may take this even further.

7.4 Further Work

We envision many possible avenues for future work.

Regarding the SSSP and PR implementations, an energy efficiency study (versus CPU and GPU) may provide more insight into viability (especially for PR) and possible optimisation avenues.

Regarding the DC1 implementation, further optimisation is likely possible. For instance: studying the incorporation of the accumulate (and, for the U optimisation, degree calculation) phases on the FPGA, possibly by adding a shared compute unit³; implementing a host-device streaming method (such as QDMA or Host Memory Access [105]) to further reduce execution times, or, if the accumulate phase is kept on the CPU, provide faster update; determining the optimal chunk size for the C optimisation during runtime.

We also believe attempting implementation of these algorithms using ThunderGP may yield interesting insights.

As for the DC1 *CSR formulation*, further study is required to determine suitability to replace BC in the NavSys flow. A large-scale study spanning several domains could also determine if DC1 is a suitable replacement for other centrality measures. The dynamic scheme can most certainly be further refined via mathematical analysis; and an iterative scheme may prove interesting if, from a graph-theoretical viewpoint, it yields useful results. Iterative execution would also necessitate rethinking both the host-side and FPGA implementation.

Acknowledgement

This work was supported in part by Xilinx and ETH Zurich under the Xilinx Adaptive Compute Cluster (XACC) program.

²In particular, HPC has its own set of common algorithms which lack FPGA HLS library implementations [48].

³There is an argument to be made, however, that the accumulate phase should be kept on the host if the user desires rapid output with live updates.

Appendix A

Distinctiveness Centrality 1 Kernel Variants

A.1 dcl

```
1 #include <hls_stream.h>
2 #include <cmath>
3
4
5 template <int ALPHA>
6 inline void dcl_stream( hls::stream<unsigned> &offset_fifo,
7                        hls::stream<unsigned> &column_fifo,
8                        hls::stream<unsigned> &weight_fifo,
9                        hls::stream<unsigned> &out_idx_fifo,
10                       hls::stream<float> &out_val_fifo,
11                       const unsigned num_vertices) {
12
13
14 setup:
15     static unsigned prev_offset = offset_fifo.read();
16     static const unsigned numer = num_vertices-1;
17
18 execute:
19     for (unsigned j = 0; j < num_vertices; ++j) {
20         const unsigned cur_offset = offset_fifo.read();
21         const unsigned cur_deg = cur_offset - prev_offset;
22
23         const float cur_deg_pow = (float) std::pow(cur_deg, ALPHA);
24         const float tree = std::log10(numer / cur_deg_pow);
25
26
27 execute_inner:
28     for (unsigned edge = 0; edge < cur_deg; ++edge) {
```

```

29 #pragma HLS PIPELINE II=1
30
31     const unsigned weight_tmp = weight_fifo.read();
32     const unsigned i_tmp = column_fifo.read();
33
34     const float value_tmp = tree * weight_tmp;
35
36     out_idx_fifo.write(i_tmp);
37     out_val_fifo.write(value_tmp);
38
39 }
40 prev_offset = cur_offset;
41 }
42 return;
43 }
44
45
46 inline void read_in_offset( const unsigned * const offset_host,
47                             hls::stream<unsigned> &offset_fifo,
48                             const unsigned num_vertices) {
49
50 mem_rd_off:
51     for (unsigned i = 0; i < num_vertices+1; ++i) {
52 #pragma HLS PIPELINE II=1
53
54         offset_fifo.write(offset_host[i]);
55     }
56 }
57
58 inline void read_in_column( const unsigned * const column_host,
59                             hls::stream<unsigned> &column_fifo,
60                             const unsigned * const weight_host,
61                             hls::stream<unsigned> &weight_fifo,
62                             const unsigned num_edges) {
63
64 mem_rd_col:
65     for (unsigned i = 0; i < num_edges; ++i) {
66 #pragma HLS PIPELINE II=1
67
68         column_fifo.write(column_host[i]);
69         weight_fifo.write(weight_host[i]);
70     }
71 }
72
73 inline void write_out( hls::stream<unsigned> &out_idx_fifo,
74                       unsigned *out_idx_host,
75                       hls::stream<float> &out_val_fifo,
76                       float *out_val_host,
77                       const unsigned num_edges) {

```

```

78
79 mem_wr:
80     for (unsigned i = 0; i < num_edges; ++i) {
81 #pragma HLS PIPELINE II=1
82
83         out_idx_host[i] = out_idx_fifo.read();
84         out_val_host[i] = out_val_fifo.read();
85     }
86 }
87
88
89 extern "C" {
90 #define ALPHA 5
91 void my_dc1(const unsigned * const offset_host,
92             const unsigned * const column_host,
93             const unsigned * const weight_host,
94             unsigned * const out_idx_host,
95             float * const out_val_host,
96             const unsigned num_vertices, const unsigned num_edges) {
97
98 #pragma HLS interface m_axi port=offset_host bundle=offset
99 #pragma HLS interface m_axi port=column_host bundle=column
100 #pragma HLS interface m_axi port=weight_host bundle=weight
101
102 #pragma HLS interface m_axi port=out_idx_host bundle=out_idx
103 #pragma HLS interface m_axi port=out_val_host bundle=out_val
104
105 static hls::stream<unsigned> offset_fifo("offset_fifo");
106 static hls::stream<unsigned> column_fifo("column_fifo");
107 static hls::stream<unsigned> weight_fifo("weight_fifo");
108
109 static hls::stream<unsigned> out_idx_fifo("out_idx_fifo");
110 static hls::stream<float> out_val_fifo("out_val_fifo");
111
112 #pragma HLS stream variable=offset_fifo depth=2
113 #pragma HLS stream variable=column_fifo depth=2
114 #pragma HLS stream variable=weight_fifo depth=2
115
116 #pragma HLS stream variable=out_idx_fifo depth=2
117 #pragma HLS stream variable=out_val_fifo depth=2
118
119
120 #pragma HLS DATAFLOW
121
122     read_in_column(    column_host, column_fifo,
123                       weight_host, weight_fifo,
124                       num_edges);
125
126     read_in_offset(    offset_host, offset_fifo,

```

```

127         num_vertices);
128
129     dc1_stream<ALPHA>( offset_fifo,
130                       column_fifo,
131                       weight_fifo,
132                       out_idx_fifo,
133                       out_val_fifo,
134                       num_vertices);
135
136     write_out(        out_idx_fifo, out_idx_host,
137                     out_val_fifo, out_val_host,
138                     num_edges);
139
140 }
141 }

```

Listing A.1: dc1 Kernel (Baseline, Full)

Table A.1: Resource usage for dc1, 1 CU, Alveo U250

Resource	Usage	Available	Usage (%)
LUT	134647	1726216	7.8001246
LUTRAM	18555	790200	2.3481398
FF	194034	3452432	5.620212
BRAM	235	2688	8.742559
DSP	122	12288	0.99283856

Table A.2: Resource usage for dc1, 1 CU, Alveo U280

Resource	Usage	Available	Usage (%)
LUT	135030	1302720	10.365236
LUTRAM	16001	600480	2.6647017
FF	182060	2607360	6.982542
BRAM	217	2016	10.763888
DSP	119	9024	1.3187057

A.2 dc1_sTable A.3: Resource usage for *dc1_s*, 4 CUs, Alveo U250

Resource	Usage	Available	Usage (%)
LUT	266473	1726216	15.436829
LUTRAM	39818	790200	5.0389776
FF	371495	3456000	10.749277
BRAM	677	2688	25.18601
DSP	473	12288	3.849284

A.3 dc1_no_idxTable A.4: Resource usage for *dc1_no_idx*, 8 CUs, Alveo U280

Resource	Usage	Available	Usage (%)
LUT	260219	1302720	19.975052
LUTRAM	47672	600480	7.938982
FF	378776	2607360	14.527185
BRAM	304	2016	15.079366
DSP	924	9024	10.239362

A.4 dc1_deg_powTable A.5: Resource usage for *dc1_deg_pow*, 4 CUs, Alveo U250

Resource	Usage	Available	Usage (%)
LUT	241993	1726216	14.018698
LUTRAM	34342	790200	4.3459883
FF	333583	3456000	9.652286
BRAM	559	2688	20.796131
DSP	137	12288	1.1149089

Table A.6: Resource usage for dc1_deg_pow, 8 CUs, Alveo U280

Resource	Usage	Available	Usage (%)
LUT	236555	1302720	18.158545
LUTRAM	42088	600480	7.009059
FF	338784	2607360	12.993373
BRAM	220	2016	10.912699
DSP	252	9024	2.7925532

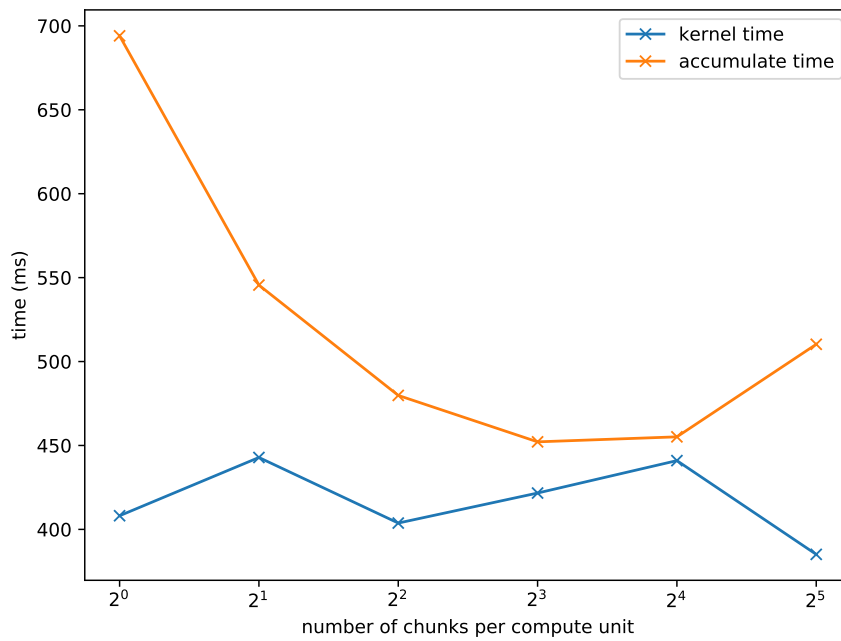


Figure A.1: Kernel Time and Total Execution Time vs. Number of Chunks per CU, for Variant DDRODPCB and Dataset CO

A.5 *dc1_no_col*

Table A.7: Resource usage for *dc1_no_col*, 10 CUs, Alveo U280

Resource	Usage	Available	Usage (%)
LUT	234274	1302720	17.98345
LUTRAM	40142	600480	6.684985
FF	319136	2607360	12.239814
BRAM	219	2016	10.863095
DSP	314	9024	3.47961

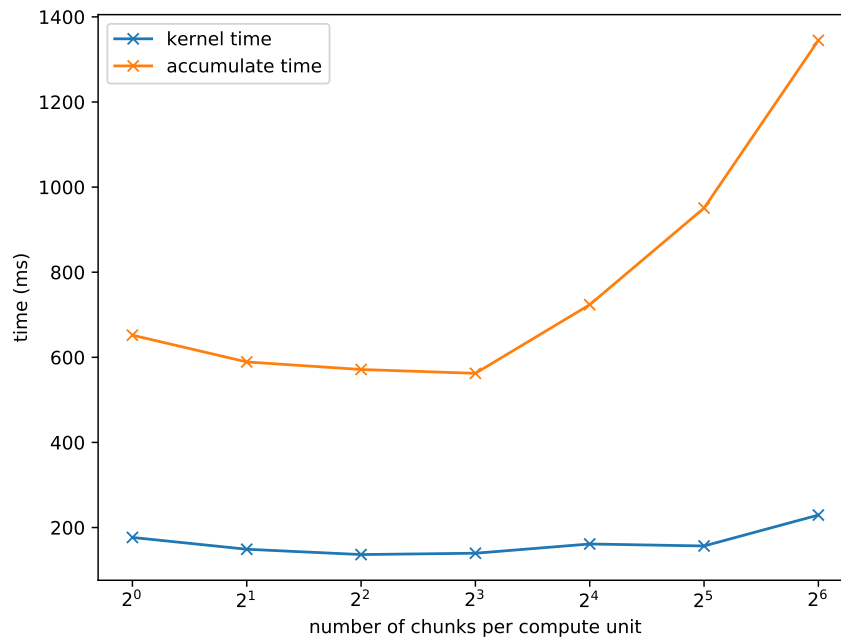


Figure A.2: Kernel Time and Total Execution Time vs. Number of Chunks per CU, for Variant DDRODPCB and Dataset CO

A.6 *dc1_unroll*

```

1 #include <hls_stream.h>
2 #include "ap_int.h"
3 #include <cmath>
4
5
6 #define PWIDTH 512
7 #define DWIDTH 32

```

```

8  #define TSIZE 16
9
10 #define GET(p,i) p.range((i + 1) * DWIDTH - 1, i * DWIDTH)
11 #define PUT(p,i,v) p.range((i + 1) * DWIDTH - 1, i * DWIDTH) = v
12
13
14 #define pint ap_uint<PWIDTH>
15 #define mint ap_uint<DWIDTH>
16
17 union f_cast {
18     float f;
19     unsigned u;
20 };
21
22 template <int ALPHA>
23 inline void dcl_stream(hls::stream<pint> &degree_fifo,
24                       hls::stream<pint> &weight_fifo,
25                       hls::stream<pint> &out_val_fifo,
26                       const unsigned total_num_vertices,
27                       const unsigned my_num_edges) {
28
29 setup:
30     static const float numer = total_num_vertices-1;
31
32 execute:
33     for (unsigned j = 0; j < (my_num_edges%TSIZE == 0 ? my_num_edges/TSIZE :
34                               my_num_edges/TSIZE + 1); j++) {
35
36 #pragma HLS PIPELINE
37
38         const pint p_deg = degree_fifo.read();
39         const pint p_weight = weight_fifo.read();
40
41         pint p_out_val = 0;
42
43         execute_inner:
44         for (unsigned i = 0; i < TSIZE; ++i) {
45 #pragma HLS UNROLL
46
47             unsigned cur_deg = GET(p_deg, i);
48
49             // Will happen at end because of padding
50             if ( cur_deg != 0 ) {
51                 float cur_deg_pow = 1.f; // larger repr range
52                 for ( int z = 0; z < ALPHA; ++z ) {
53                     #pragma HLS UNROLL
54                     cur_deg_pow *= cur_deg;
55                 }
56
57                 const float tree = std::log10(numer / cur_deg_pow);

```

```

56         const unsigned weight_tmp = GET(p_weight, i);
57
58         f_cast value_tmp;
59         value_tmp.f = tree * weight_tmp;
60
61         PUT(p_out_val, i, value_tmp.u);
62
63     }
64
65     else PUT(p_out_val, i, 0);
66 }
67 out_val_fifo.write(p_out_val);
68 }
69 return;
70 }
71
72
73 inline void read_in_column(const pint * const degree_host, hls::stream<pint> &
    degree_fifo,
74                         const pint * const weight_host, hls::stream<pint> &weight_fifo,
75                         const unsigned my_num_edges) {
76 mem_rd_col:
77     for (unsigned i = 0; i < (my_num_edges%TSIZE == 0 ? my_num_edges/TSIZE :
    my_num_edges/TSIZE + 1); i++) {
78 #pragma HLS PIPELINE II=1
79         degree_fifo.write(degree_host[i]);
80         weight_fifo.write(weight_host[i]);
81     }
82 }
83
84 inline void write_out(hls::stream<pint> &out_val_fifo, pint *out_val_host,
85                     const unsigned my_num_edges) {
86 mem_wr:
87     for (unsigned i = 0; i < (my_num_edges%TSIZE == 0 ? my_num_edges/TSIZE :
    my_num_edges/TSIZE + 1); i++) {
88 #pragma HLS PIPELINE II=1
89         out_val_host[i] = out_val_fifo.read();
90     }
91 }
92
93 extern "C" {
94 #define ALPHA 5
95
96 void my_dcl_unroll(const pint * const degree_host,
97                  const pint * const weight_host,
98                  pint * const out_val_host,
99                  const unsigned total_num_vertices,
100                  const unsigned my_num_vertices, const unsigned my_num_edges) {
101

```

```

102 #pragma HLS interface m_axi port=degree_host bundle=degree
103 #pragma HLS interface m_axi port=weight_host bundle=weight
104 #pragma HLS interface m_axi port=out_val_host bundle=out_val
105 #pragma HLS interface ap_ctrl_chain port=return
106
107
108     static hls::stream<pint>    degree_fifo("degree_fifo");
109     static hls::stream<pint>    weight_fifo("weight_fifo");
110
111     static hls::stream<pint>    out_val_fifo("out_val_fifo");
112
113
114 #pragma HLS stream variable=degree_fifo depth=2
115 #pragma HLS stream variable=weight_fifo depth=3
116
117 #pragma HLS stream variable=out_val_fifo depth=2
118
119
120 #pragma HLS DATAFLOW
121
122     read_in_column(    degree_host, degree_fifo,
123                       weight_host, weight_fifo,
124                       my_num_edges);
125
126     dc1_stream<ALPHA>( degree_fifo,
127                       weight_fifo,
128                       out_val_fifo,
129                       total_num_vertices,
130                       my_num_edges);
131
132     write_out(        out_val_fifo, out_val_host,
133                       my_num_edges);
134 }
135 }

```

Listing A.2: dc1_unroll Kernel (Full)

Table A.8: Resource usage for dc1_unroll, 4 CUs, Alveo U250

Resource	Usage	Available	Usage (%)
LUT	383561	1726216	22.219757
LUTRAM	50078	790200	6.337383
FF	542991	3456000	15.711544
BRAM	569	2688	21.168154
DSP	1997	12288	16.251629

Table A.9: Resource usage for dcl_unroll, 10 CUs, Alveo U280

Resource	Usage	Available	Usage (%)
LUT	635953	1302720	48.81732
LUTRAM	101448	600480	16.894485
FF	956862	2607360	36.698498
BRAM	434	2016	21.527777
DSP	4964	9024	55.008865

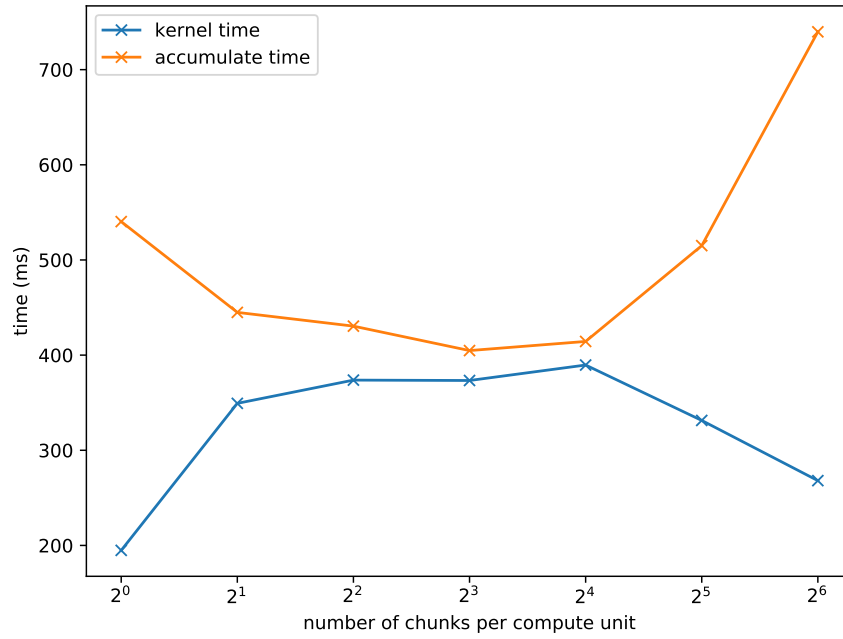


Figure A.3: Kernel Time and Total Execution Time Excluding Degree Queue Generation vs. Number of Chunks per CU, for Variant DDRODPUCB and Dataset CO

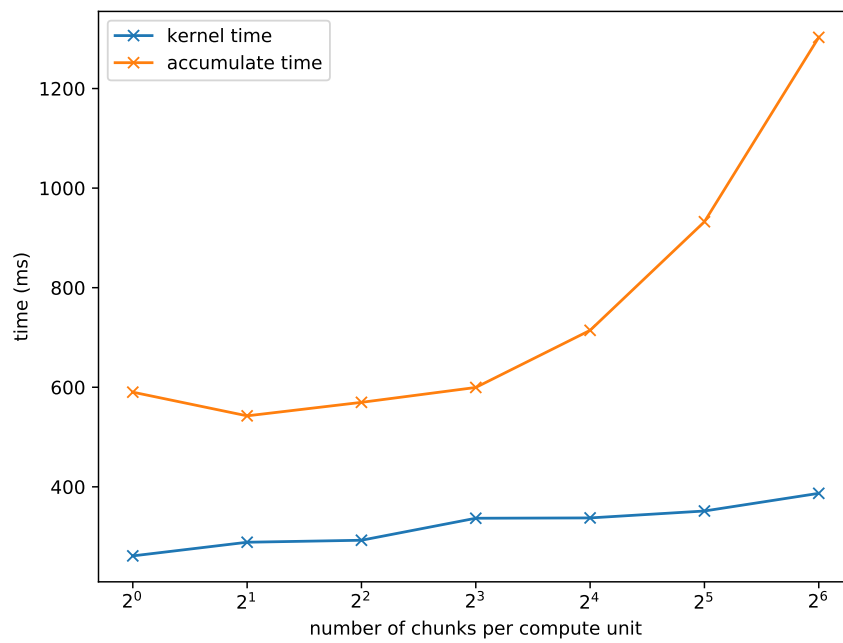


Figure A.4: Kernel Time and Total Execution Time Excluding Degree Queue Generation vs. Number of Chunks per CU, for Variant DDRODPUCB and Dataset CO

Appendix B

Absolute Execution Metrics

B.1 SSSP

Table B.1: Execution Metrics for SSSP

Dataset	Total Time	Write Time	Kernel Time	Read Time	Source
R4	31990	35.596 ± 2	31952 ± 0.6	2.9847 ± 0.1	0
RMAT	17260	31.964 ± 2	17226 ± 0.3	1.5275 ± 0.03	0
SO	24738	100.98 ± 7	24636 ± 0.4	1.0956 ± 0.03	0
LJ	25644	68.467 ± 5	25573 ± 1	1.7202 ± 0.02	0

B.2 PR

Table B.2: Execution Metrics for PR-DDR

Dataset	Total Time	Write Time	Kernel Time	Read Time	Iterations	ϵ
R4	40614	48.699 ± 1	40560 ± 10	5.418 ± 0.2	31	0.01
RMAT	59320	37.861 ± 2	59279 ± 40	2.7927 ± 0.05	46	0.01
SO	106770	218.17 ± 8	106550 ± 10	2.0029 ± 0.06	20	0.1
LJ	35921	89.729 ± 1	35828 ± 4	3.1979 ± 0.1	20	0.1
CO	76156	236.06 ± 8	75917 ± 0.6	2.1694 ± 0.05	5	0.01

Table B.3: Execution Metrics for PR-HBM2C

Dataset	Total Time	Write Time	Kernel Time	Read Time	Iterations	ϵ
R4	24395	43.861 ± 0.6	24344 ± 0.1	7.3767 ± 0.3	31	0.01
RMAT	30705	34.455 ± 0.3	30667 ± 0.4	3.8085 ± 0.07	46	0.01
LJ	30184	78.27 ± 0.7	30102 ± 0.2	4.4179 ± 0.2	20	0.1

Table B.4: Execution Metrics for PR-HBM6C

Dataset	Total Time	Write Time	Kernel Time	Read Time	Iterations	ϵ
R4	33788	45.993 ± 0.4	33735 ± 0.06	7.164 ± 0.5	31	0.01
RMAT	38664	37.776 ± 0.3	38623 ± 0.1	3.5975 ± 0.2	46	0.01
LJ	49814	86.768 ± 0.6	49723 ± 0.1	4.1999 ± 0.1	20	0.1

Table B.5: Execution Metrics for PR-HBMJ

Dataset	Total Time	Write Time	Kernel Time	Read Time	Iterations	ϵ
R4	27442	48.547 ± 0.1	27387 ± 0.2	7.1121 ± 0.4	31	0.01
RMAT	33980	41.122 ± 0.3	33935 ± 0.3	3.7868 ± 0.2	46	0.01
SO	95816	225.34 ± 0.3	95588 ± 0.3	2.7326 ± 0.2	20	0.1
LJ	34127	96.715 ± 0.3	34026 ± 0.1	4.3387 ± 0.1	20	0.1

B.3 DC1

Table B.6: Execution Metrics for DDR1

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	4178.6	177.12 ± 6	33.913 ± 2	3943.6 ± 0.8	24.026 ± 0.9
RMAT	2184.2	128 ± 7	32.745 ± 2	1999.5 ± 0.3	24.029 ± 0.5
US	11309	74.841 ± 2	64.755 ± 2	11129 ± 0.8	40.839 ± 2
SO	2054.5	156.61 ± 6	105.62 ± 6	1715.7 ± 0.03	76.564 ± 2
LJ	2682.5	138.98 ± 4	69.933 ± 3	2423.6 ± 0.02	49.955 ± 2
CO	3052	445.85 ± 20	242.34 ± 10	2189.2 ± 0.2	174.63 ± 4

Table B.7: Execution Metrics for DDR4

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	1100.8	74.967 ± 2	34.316 ± 0.7	1011.2 ± 0.7	31.276 ± 0.9
RMAT	655.9	84.009 ± 3	33.555 ± 0.7	557.51 ± 0.7	74.474 ± 0.7
US	2931.2	51.2 ± 2	61.183 ± 2	2853.4 ± 2	104.06 ± 2
SO	1159.6	172.1 ± 3	109.56 ± 2	940.22 ± 2	725.88 ± 3
LJ	1663.3	106 ± 2	65.09 ± 3	1528.4 ± 3	1335.3 ± 2
CO	1434.1	375.32 ± 8	239.61 ± 8	955.54 ± 7	548.87 ± 8

Table B.8: Execution Metrics for DDR4ODP

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	158.98	71.669 ± 3	34.368 ± 0.6	74.465 ± 0.4	29.227 ± 0.4
RMAT	157.32	81.996 ± 1	32.677 ± 1	63.28 ± 1	29.295 ± 1
US	229.03	51.049 ± 2	62.325 ± 2	153.92 ± 1	53.172 ± 1
SO	404.43	170.57 ± 3	105.48 ± 5	193.71 ± 5	95.756 ± 4
LJ	270.32	102.63 ± 3	69.836 ± 1	141.77 ± 0.8	78.717 ± 0.8
CO	882.02	370.88 ± 4	237.97 ± 9	420.67 ± 7	205.19 ± 8

Table B.9: Execution Metrics for DDR4ODPCB1

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	118.98	118.98 ± 2	34.561 ± 2	74.506 ± 2	33.797 ± 2
RMAT	116.02	116.02 ± 1	31.735 ± 2	61.611 ± 1	36.703 ± 2
US	184.91	184.91 ± 2	58.281 ± 2	149.9 ± 2	49.992 ± 0.7
SO	312.17	312.17 ± 3	108.45 ± 6	193.01 ± 7	117.07 ± 9
LJ	196.79	196.79 ± 3	69.537 ± 2	122.8 ± 2	69.163 ± 8
CO	680.83	680.83 ± 9	206.97 ± 3	396.31 ± 3	245.86 ± 6

Table B.10: Execution Metrics for DDR4ODPCUB1

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	95.892	95.892 ± 5	54.214 ± 4	51.935 ± 4	51.939 ± 4
RMAT	97.083	97.083 ± 4	51.834 ± 8	48.918 ± 9	50.716 ± 6
US	102.89	102.89 ± 2	58.756 ± 1	54.733 ± 3	50.615 ± 4
SO	253.65	253.65 ± 5	138.49 ± 20	131.24 ± 20	135.39 ± 9
LJ	151.96	151.96 ± 6	81.951 ± 3	73.97 ± 7	75.555 ± 4
CO	551.44	551.44 ± 5	317.81 ± 9	301.11 ± 9	303.04 ± 6

Table B.11: Execution Metrics for DDR4ODPCUB1, Excluding Degree Calculation

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	86.614	86.614 ± 6	52.14 ± 8	49.739 ± 7	51.164 ± 6
RMAT	87.381	87.381 ± 3	51.659 ± 4	48.825 ± 6	50.279 ± 5
US	85.777	85.777 ± 2	55.519 ± 5	50.644 ± 8	47.642 ± 6
SO	232.9	232.9 ± 10	137.1 ± 20	124.48 ± 20	129.22 ± 20
LJ	137.36	137.36 ± 5	74.969 ± 6	65.46 ± 10	70.275 ± 5
CO	511.15	511.15 ± 20	310.54 ± 30	292.91 ± 30	299.13 ± 20

Table B.12: Execution Metrics for HBM1J7

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	4129.6	127.48 ± 10	31.5 ± 0.3	3943 ± 0.06	27.584 ± 1
RMAT	2204.6	120.67 ± 9	29.932 ± 0.2	2025.9 ± 0.06	28.132 ± 2
US	11277	42.779 ± 1	57.122 ± 0.4	11129 ± 0.03	48.563 ± 2
SO	2170.2	257.93 ± 10	93.36 ± 0.4	1725.2 ± 0.03	93.683 ± 4
LJ	2740.4	178.89 ± 8	60.696 ± 0.1	2442.2 ± 0.05	58.602 ± 0.5
CO	3164.5	567.4 ± 10	207.54 ± 0.6	2187.1 ± 0.03	202.48 ± 3

Table B.13: Execution Metrics for HBM8

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	632.12	105.33 ± 2	31.669 ± 0.3	520.83 ± 0.3	29.657 ± 0.3
RMAT	416.6	114.46 ± 3	30.244 ± 0.3	296.65 ± 0.3	52.734 ± 0.3
US	1528.5	50.77 ± 7	58.872 ± 0.7	1466.6 ± 0.7	94.532 ± 0.7
SO	1000.9	272.31 ± 5	91.011 ± 0.3	712.74 ± 0.3	624.15 ± 0.3
LJ	1263.5	167.26 ± 6	60.525 ± 0.4	1085.6 ± 0.4	1001.9 ± 0.5
CO	1210.7	572.71 ± 4	196.88 ± 0.6	604.16 ± 0.7	412.14 ± 0.7

Table B.14: Execution Metrics for HBM8ODP

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	174.34	116.02 ± 1	31.707 ± 0.6	52.229 ± 0.6	29.181 ± 0.6
RMAT	170.88	119.43 ± 2	30.023 ± 0.4	45.932 ± 0.3	28.518 ± 0.3
US	179.14	62.851 ± 1	58.657 ± 1	105.26 ± 1	53.608 ± 1
SO	431.14	278.14 ± 5	90.576 ± 0.1	137.19 ± 0.1	91.629 ± 0.2
LJ	281.84	167.75 ± 5	59.896 ± 0.5	103.6 ± 0.5	74.328 ± 0.5
CO	895.49	579.02 ± 20	196.38 ± 0.6	282.73 ± 0.6	182.21 ± 0.6

Table B.15: Execution Metrics for HBM10ODP

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	141.26	102.57 ± 3	17.048 ± 0.7	34.11 ± 0.6	16.05 ± 0.9
RMAT	149.6	116.36 ± 7	15.434 ± 0.7	28.558 ± 0.6	15.14 ± 1
US	124.42	45.433 ± 3	32.732 ± 0.4	72.367 ± 0.4	30.846 ± 0.4
SO	344.34	244.67 ± 3	48.615 ± 0.7	86.912 ± 0.5	50.542 ± 0.5
LJ	236.65	159.52 ± 4	32.053 ± 0.5	68.702 ± 0.3	45.318 ± 0.3
CO	787.24	571.84 ± 9	107.42 ± 0.6	176.28 ± 0.6	107.69 ± 1

Table B.16: Execution Metrics for HBM10ODPCB1

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	116.06	116.06 ± 3	18.741 ± 0.5	35.647 ± 0.5	18.815 ± 2
RMAT	127.25	127.25 ± 4	17.418 ± 1	29.414 ± 0.8	20.342 ± 7
US	104.45	104.45 ± 4	38.652 ± 0.4	76.022 ± 3	34.879 ± 3
SO	294.18	294.18 ± 4	53.318 ± 2	84.812 ± 5	53.238 ± 3
LJ	181.44	181.44 ± 5	34.809 ± 2	57.701 ± 4	38.266 ± 7
CO	656.24	656.24 ± 9	120.64 ± 9	182.85 ± 8	113.14 ± 7

Table B.17: Execution Metrics for HBM10ODPUCB1

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	119.53	119.53 ± 2	45.54 ± 3	43.695 ± 3	44.776 ± 4
RMAT	130.81	130.81 ± 3	43.511 ± 3	40.834 ± 3	40.986 ± 2
US	98.891	98.891 ± 10	73.925 ± 3	69.612 ± 3	69.617 ± 3
SO	274.21	274.21 ± 9	138.3 ± 10	129.54 ± 10	129.98 ± 10
LJ	177.6	177.6 ± 6	90.202 ± 8	84.902 ± 9	85.553 ± 8
CO	630.69	630.69 ± 30	310.47 ± 30	282.85 ± 10	282.12 ± 10

Table B.18: Execution Metrics for HBM10ODPUCB1, Excluding Degree Calculation

Dataset	Total Time	Accumulate Time	Write Time	Kernel Time	Read Time
R4	113.01	113.01 ± 7	47.281 ± 6	41.005 ± 6	42.948 ± 6
RMAT	121.21	121.21 ± 3	49.672 ± 8	45.61 ± 5	45.91 ± 6
US	92.772	92.772 ± 6	74.376 ± 4	68.801 ± 3	68.529 ± 3
SO	283.15	283.15 ± 10	142.65 ± 10	126.07 ± 10	126.66 ± 10
LJ	170.21	170.21 ± 8	93.552 ± 6	84.483 ± 5	84.101 ± 5
CO	581.63	581.63 ± 7	299.24 ± 10	280.71 ± 10	279.52 ± 10

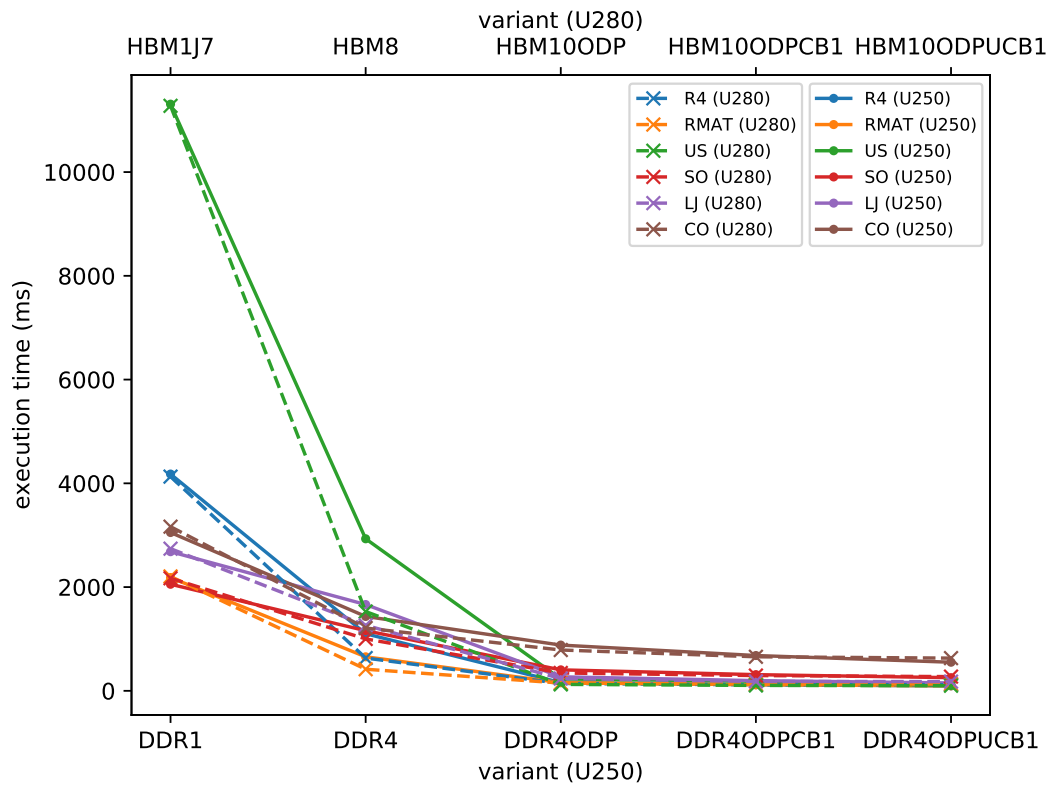


Figure B.1: Total Execution Time for all DDR1 Variants, in Progression Order

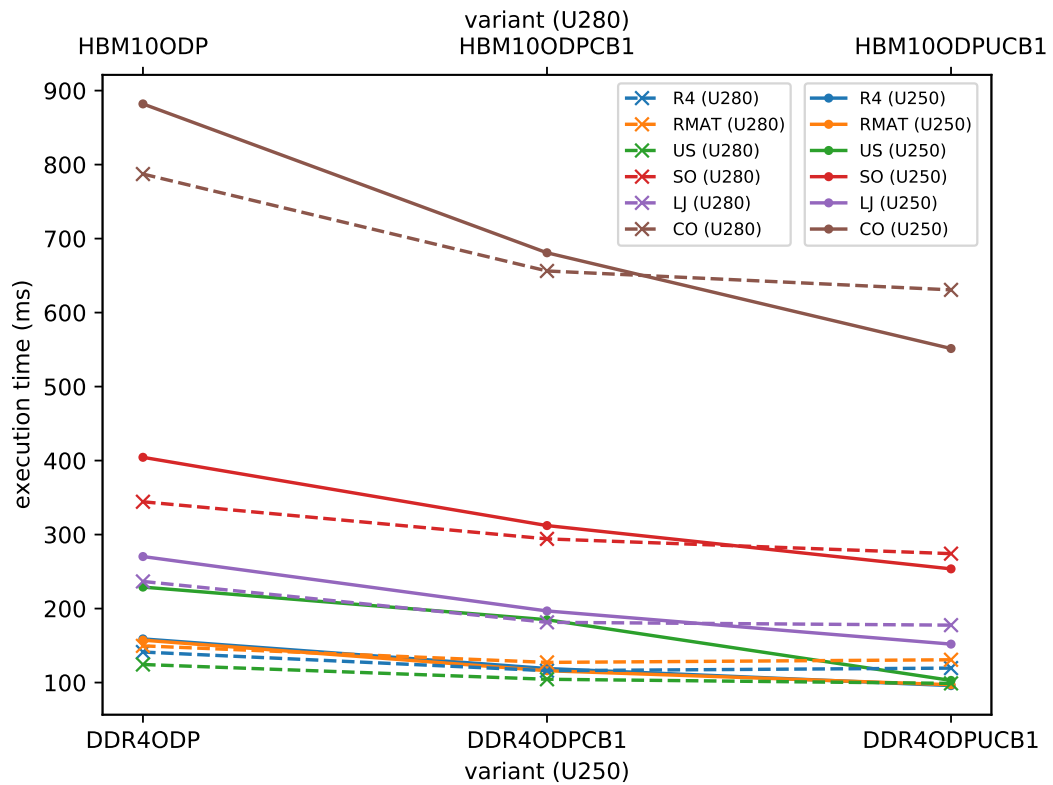


Figure B.2: Total Execution Time for DDR1 D Optimisation Variants, in Progression Order

References

- [1] Peter Alfke, Ivo Bolsens, Bill Carter, Mike Santarini, and Steve Trimberger. It’s an FPGA! *IEEE Solid-State Circuits Magazine*, 3(4):15–20, 23. DOI: 10.1109/MSSC.2011.942449. URL: <http://ieeexplore.ieee.org/document/6069771/>.
- [2] AMD. AMD to Acquire Xilinx, Creating the Industry’s High Performance Computing Leader, October 2020. URL: <https://www.amd.com/en/press-releases/2020-10-27-amd-to-acquire-xilinx-creating-the-industry-s-high-performance-computing>.
- [3] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering*, volume 9220, pages 19–80. Springer International Publishing, Cham, 2016. DOI: 10.1007/978-3-319-49487-6_2. URL: http://link.springer.com/10.1007/978-3-319-49487-6_2.
- [4] Reinhard Bauer and Dorothea Wagner. Batch Dynamic Single-Source Shortest-Path Algorithms: An Experimental Study. In Jan Vahrenhold, editor, *Experimental Algorithms*, volume 5526, pages 51–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. DOI: 10.1007/978-3-642-02011-7_7. URL: http://link.springer.com/10.1007/978-3-642-02011-7_7.
- [5] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. Graph Processing on FPGAs: Taxonomy, Survey, Challenges. *arXiv:1903.06697 [cs]*, April 2019. URL: <http://arxiv.org/abs/1903.06697>.
- [6] João Bispo, Jorge G Barbosa, Pedro Filipe Silva, Cristian Morales, Mirko Myllykoski, Pedro Ojeda-May, Milosz Bialczak, Mariusz Uchonski, Adam Wlodarczyk, Peter Wauligmann, et al. Best Practice Guide: Modern Accelerators. Technical report, PRACE aisbl, June 2021. URL: <https://prace-ri.eu/training-support/best-practice-guides/modern-accelerators/>.
- [7] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Compiling Graph Applications for GPU s with GraphIt. In *2021 IEEE/ACM Int. Symp. Code Gener. Optim.*, pages 248–261, Seoul, Korea (South), February 2021. IEEE. DOI: 10.1109/CGO51591.2021.9370321. URL: <https://ieeexplore.ieee.org/document/9370321/>.
- [8] Ulrik Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2):136–145, May 2008. DOI: 10.1016/j.socnet.2007.11.001. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0378873307000731>.

- [9] Nick Brown. Weighing Up the New Kid on the Block: Impressions of using Vitis for HPC Software Development. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 335–340, Gothenburg, Sweden, August 2020. IEEE. DOI: 10.1109/FPL50879.2020.00062. URL: <https://ieeexplore.ieee.org/document/9221613/>.
- [10] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637, September 2018. DOI: 10.1109/TKDE.2018.2807452. URL: <https://ieeexplore.ieee.org/document/8294302/>.
- [11] Xinyu Chen, Ronak Bajaj, Yao Chen, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. On-The-Fly Parallel Data Shuffling for Graph Processing on OpenCL-Based FPGAs. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 67–73, Barcelona, Spain, September 2019. IEEE. DOI: 10.1109/FPL.2019.00020. URL: <https://ieeexplore.ieee.org/document/8892034/>.
- [12] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. ThunderGP: HLS-based Graph Processing Framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 69–80, Virtual Event USA, February 2021. ACM. DOI: 10.1145/3431920.3439290. URL: <https://dl.acm.org/doi/10.1145/3431920.3439290>.
- [13] Claudio Rocchini. Betweenness Centrality, 2007. URL: https://commons.wikimedia.org/wiki/File:Graph_betweenness.svg.
- [14] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011. DOI: 10.1109/TCAD.2011.2110592. URL: <http://ieeexplore.ieee.org/document/5737854/>.
- [15] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shao-chong Zhang. Understanding Performance Differences of FPGAs and GPUs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, Boulder, CO, USA, April 2018. IEEE. DOI: 10.1109/FCCM.2018.00023. URL: <https://ieeexplore.ieee.org/document/8457638/>.
- [16] P. Coussy, D.D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, July 2009. DOI: 10.1109/MDT.2009.69. URL: <http://ieeexplore.ieee.org/document/5209958/>.
- [17] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 105–110, Monterey California USA, February 2016. ACM. DOI: 10.1145/2847263.2847339. URL: <https://dl.acm.org/doi/10.1145/2847263.2847339>.

- [18] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pages 217–226, Monterey, California, USA, 2017. ACM Press. DOI: 10.1145/3020078.3021739. URL: <http://dl.acm.org/citation.cfm?doid=3020078.3021739>.
- [19] Alessandro D’Alconzo, Idilio Drago, Andrea Morichetta, Marco Mellia, and Pedro Casas. A Survey on Big Data for Network Traffic Monitoring and Analysis. *IEEE Transactions on Network and Service Management*, 16(3):800–813, September 2019. DOI: 10.1109/TNSM.2019.2933358. URL: <https://ieeexplore.ieee.org/document/8789667/>.
- [20] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In Mikhail J. Atallah and Marina Blanton, editors, *Algorithms and Theory of Computation Handbook: General Concepts and Techniques*, Chapman & Hall/CRC Applied Algorithms and Data Structures Series. Taylor & Francis, Boca Raton, Fla, 2nd ed edition, 2009. URL: <https://dl.acm.org/doi/10.5555/1882757.1882766>.
- [21] DIMACS. 9th DIMACS Implementation Challenge - Shortest Paths, 2006. URL: <http://www.diag.uniroma1.it/challenge9/>.
- [22] Stefan Edelkamp and Stefan Schrödl. Basic Search Algorithms. In *Heuristic Search*, pages 47–87. Elsevier, 2012. DOI: 10.1016/B978-0-12-372512-7.00002-X. URL: <https://linkinghub.elsevier.com/retrieve/pii/B978012372512700002X>.
- [23] Nina Engelhardt and Hayden K.-H. So. GraVF-M: Graph Processing System Generation for Multi-FPGA Platforms. *ACM Transactions on Reconfigurable Technology and Systems*, 12(4):1–28, November 2019. DOI: 10.1145/3357596. URL: <https://dl.acm.org/doi/10.1145/3357596>.
- [24] Nina Engelhardt and Hayden Kwok-Hay So. GraVF: A vertex-centric distributed graph processing framework on FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Lausanne, Switzerland, August 2016. IEEE. DOI: 10.1109/FPL.2016.7577360. URL: <http://ieeexplore.ieee.org/document/7577360/>.
- [25] Kermin Fleming and Michael Adler. The LEAP FPGA Operating System. In Dirk Koch, Frank Hannig, and Daniel Ziener, editors, *FPGAs for Software Programmers*, pages 245–258. Springer International Publishing, Cham, 2016. DOI: 10.1007/978-3-319-26408-0_14. URL: http://link.springer.com/10.1007/978-3-319-26408-0_14.
- [26] Massimo Franceschet. PageRank: Standing on the shoulders of giants. *Communications of the ACM*, 54(6):92–101, June 2011. DOI: 10.1145/1953122.1953146. URL: <https://dl.acm.org/doi/10.1145/1953122.1953146>.
- [27] Andrea Fronzetti Colladon and Maurizio Naldi. Distinctiveness centrality in social networks. *PLOS ONE*, 15(5):e0233276, May 2020. DOI: 10.1371/journal.pone.0233276. URL: <https://dx.plos.org/10.1371/journal.pone.0233276>.
- [28] Quentin Gautier, Alric Althoff, Pingfan Meng, and Ryan Kastner. Spector: An OpenCL FPGA benchmark suite. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 141–148, Xi’an, China, December 2016. IEEE.

- DOI: 10.1109/FPT.2016.7929519. URL: <http://ieeexplore.ieee.org/document/7929519/>.
- [29] David F. Gleich. PageRank Beyond the Web. *SIAM Review*, 57(3):321–363, January 2015. DOI: 10.1137/140976649. URL: <http://epubs.siam.org/doi/10.1137/140976649>.
- [30] Atharva Gondhalekar and Wu-Chun Feng. Exploring FPGA Optimizations in OpenCL for Breadth-First Search on Sparse Graph Datasets. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 133–137, Gothenburg, Sweden, August 2020. IEEE. DOI: 10.1109/FPL50879.2020.00032. URL: <https://ieeexplore.ieee.org/document/9221543/>.
- [31] Rajesh Gupta and Forrest Brewer. High-Level Synthesis: A Retrospective. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 13–28. Springer Netherlands, Dordrecht, 2008. DOI: 10.1007/978-1-4020-8588-8_2. URL: http://link.springer.com/10.1007/978-1-4020-8588-8_2.
- [32] J. Hellendoorn, B. De Schutter, L.D. Baskar, and Z. Papp. Traffic control and intelligent vehicle highway systems: A survey. *IET Intelligent Transport Systems*, 5(1):38–52, March 2011. DOI: 10.1049/iet-its.2009.0001. URL: <https://digital-library.theiet.org/content/journals/10.1049/iet-its.2009.0001>.
- [33] Mohammad Hosseinabady and Jose Luis Nunez-Yanez. A Streaming Dataflow Engine for Sparse Matrix-Vector Multiplication Using High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(6):1272–1285, June 2020. DOI: 10.1109/TCAD.2019.2912923. URL: <https://ieeexplore.ieee.org/document/8695747/>.
- [34] Intel. FPGA IP (Intellectual Property) Cores. URL: <https://www.intel.com/content/www/us/en/products/programmable/intellectual-property.html>.
- [35] Intel. Intel oneAPI Toolkits. URL: <https://software.intel.com/content/www/us/en/develop/tools/oneapi/all-toolkits.html>.
- [36] Intel. Intel Ships Stratix 10 DX FPGAs; VMware Among Early Partners, September 2019. URL: <https://newsroom.intel.com/news/intel-ships-stratix-10-dx-fpgas-vmware-among-early-partners/>.
- [37] Intel. Intel High Level Synthesis Compiler Pro Edition Reference Manual. Technical Report MNL-1083, Intel, June 2021. URL: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/mnl-hls-reference.pdf>.
- [38] Intel. Intel High Level Synthesis Compiler Pro Edition: User Guide. Technical Report UG-20037, Intel, June 2021. URL: <https://www.intel.com/content/www/us/en/programmable/documentation/ewa1457708982563.html>.
- [39] Bill Jenkins. Intro to FPGA Flows Overview, May 2019. URL: https://www.lrz.de/services/compute/courses/x_lecturenotes/IntroFPGAFlows.pdf.
- [40] João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. *Embedded Computing for High Performance*. Elsevier, 2017. DOI: 10.1016/C2015-0-00283-0. URL: <https://linkinghub.elsevier.com/retrieve/pii/C20150002830>.

- [41] 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, February 2018. DOI: 10.1109/TKDE.2017.2762294. URL: <https://ieeexplore.ieee.org/document/8066363/>.
- [42] Nachiket Kapre. Custom FPGA-based soft-processors for sparse graph acceleration. In *2015 IEEE 26th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 9–16, Toronto, ON, Canada, July 2015. IEEE. DOI: 10.1109/ASAP.2015.7245698. URL: <http://ieeexplore.ieee.org/document/7245698/>.
- [43] Elias Koromilas, Ioannis Stamelos, Christoforos Kachris, and Dimitrios Soudris. Spark acceleration on FPGAs: A use case on machine learning in Pynq. In *2017 6th International Conference on Modern Circuits and Systems Technologies (MOCASST)*, pages 1–4, Thessaloniki, Greece, May 2017. IEEE. DOI: 10.1109/MOCASST.2017.7937637. URL: <http://ieeexplore.ieee.org/document/7937637/>.
- [44] Charalampos Kritikakis and Dirk Koch. End-to-end Dynamic Stream Processing on Maxeler HLS Platforms. In *2019 IEEE 30th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 59–66, New York, NY, USA, July 2019. IEEE. DOI: 10.1109/ASAP.2019.00-29. URL: <https://ieeexplore.ieee.org/document/8825019/>.
- [45] Konstantinos Krommydas, Wu-chun Feng, Christos D. Antonopoulos, and Nikolaos Bellas. OpenDwarfs: Characterization of Dwarf-Based Benchmarks on Fixed and Reconfigurable Architectures. *Journal of Signal Processing Systems*, 85(3):373–392, December 2016. DOI: 10.1007/s11265-015-1051-z. URL: <http://link.springer.com/10.1007/s11265-015-1051-z>.
- [46] Ian Kuon, Russell Tessier, and Jonathan Rose. FPGA Architecture: Survey and Challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2):135–253, 2007. DOI: 10.1561/10000000005. URL: <http://www.nowpublishers.com/article/Details/EDA-005>.
- [47] Sakari Lahti, Panu Sjoval, Jarno Vanne, and Timo D. Hamalainen. Are We There Yet? a Study on the State of High-Level Synthesis. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 38(5):898–911, May 2019. DOI: 10.1109/TCAD.2018.2834439. URL: <https://ieeexplore.ieee.org/document/8356004/>.
- [48] Joshua Lant, Javier Navaridas, Mikel Lujan, and John Goodacre. Toward FPGA-Based HPC: Advancing Interconnect Technologies. *IEEE Micro*, 40(1):25–34, January 2020. DOI: 10.1109/MM.2019.2950655. URL: <https://ieeexplore.ieee.org/document/8889391/>.
- [49] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014. URL: <http://snap.stanford.edu/data>.
- [50] Xiangwei Li, Abhishek Kumar Jain, Douglas L. Maskell, and Suhaib A. Fahmy. A time-multiplexed FPGA overlay with linear interconnect. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1075–1080, Dresden, March 2018. IEEE. DOI: 10.23919/DATE.2018.8342171. URL: <https://ieeexplore.ieee.org/document/8342171/>.

- [51] Cheng Liu, Xinyu Chen, Bingsheng He, Xiaofei Liao, Ying Wang, and Lei Zhang. OBFS: OpenCL Based BFS Optimizations on Software Programmable FPGAs. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 315–318, Tianjin, China, December 2019. IEEE. DOI: 10.1109/ICFPT47387.2019.00056. URL: <https://ieeexplore.ieee.org/document/8977884/>.
- [52] Niels Lohmann. JSON for Modern C++, 2021. URL: <https://github.com/nlohmann>.
- [53] G. Martin and G. Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers*, 26(4):18–25, July 2009. DOI: 10.1109/MDT.2009.83. URL: <https://ieeexplore.ieee.org/document/5209959/>.
- [54] John Matta, Gunes Ercal, and Koushik Sinha. Comparing the speed and accuracy of approaches to betweenness centrality approximation. *Computational Social Networks*, 6(1):2, December 2019. DOI: 10.1186/s40649-019-0062-5. URL: <https://computational-social-networks.springeropen.com/articles/10.1186/s40649-019-0062-5>.
- [55] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. A pattern based algorithmic autotuner for graph processing on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 201–213, Washington District of Columbia, February 2019. ACM. DOI: 10.1145/3293883.3295716. URL: <https://dl.acm.org/doi/10.1145/3293883.3295716>.
- [56] Kevin Morris. HLS versus OpenCL. *Electronic Engineering Journal*, March 2013. URL: <https://www.eejournal.com/article/20130312-highlevel/>.
- [57] Fahad Bin Muslim, Liang Ma, Mehdi Roozmeh, and Luciano Lavagno. Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis. *IEEE Access*, 5:2747–2762, 2017. DOI: 10.1109/ACCESS.2017.2671881. URL: <https://ieeexplore.ieee.org/document/7859319/>.
- [58] Attila M. Nagy and Vilmos Simon. Survey on traffic prediction in smart cities. *Pervasive and Mobile Computing*, 50:148–163, October 2018. DOI: 10.1016/j.pmcj.2018.07.004. URL: <https://linkinghub.elsevier.com/retrieve/pii/S1574119217306521>.
- [59] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, October 2016. DOI: 10.1109/TCAD.2015.2513673. URL: <https://ieeexplore.ieee.org/document/7368920/>.
- [60] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A Compiler Infrastructure for Accelerator Generators. *arXiv:2102.09713 [cs]*, February 2021. URL: <http://arxiv.org/abs/2102.09713>.
- [61] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *Proc. 26th ACM Intl. Conf. Arch. Supp. Prog. Lang. Oper. Syst.*, pages 804–817, Virtual USA, April 2021. ACM. DOI: 10.1145/3445814.3446712. URL: <https://dl.acm.org/doi/10.1145/3445814.3446712>.

- [62] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, Jose F. Martinez, 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, Boston, MA, USA, May 2014. IEEE. DOI: 10.1109/FCCM.2014.15. URL: <http://ieeexplore.ieee.org/document/6861577/>.
- [63] Tayo Oguntebi and Kunle Olukotun. GraphOps: A Dataflow Library for Graph Analytics Acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 111–117, Monterey California USA, February 2016. ACM. DOI: 10.1145/2847263.2847337. URL: <https://dl.acm.org/doi/10.1145/2847263.2847337>.
- [64] Keshav Pingali. High-speed graph analytics with the galois system. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications - PPAA '14*, pages 41–42, Orlando, Florida, USA, 2014. ACM Press. DOI: 10.1145/2567634.2567648. URL: <http://dl.acm.org/citation.cfm?doid=2567634.2567648>.
- [65] Pedro Pinto, João Bispo, João Cardoso, Jorge Gomes Barbosa, Davide Gadioli, Gianluca Palermo, Jan Martinovič, Martin Golasowski, Kateřina Slaninová, Radim Cmar, and Cristina Silvano. Pegasus: Performance Engineering for Software Applications Targeting HPC Systems. *IEEE Transactions on Software Engineering*, 2020. DOI: 10.1109/TSE.2020.3001257. URL: <https://ieeexplore.ieee.org/document/9113456/>.
- [66] Chris Riley. Basic Tutorial for Maximizing Memory Bandwidth with Vitis and Xilinx UltraScale+ HBM Devices, November 2019. URL: <https://www.xilinx.com/developer/articles/maximizing-memory-bandwidth-with-vitis-and-xilinx-ultrascale-hbm-devices.html>.
- [67] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization, 2015. URL: <https://networkrepository.com>.
- [68] Matthew Roughan and Jonathan Tuke. Unravelling Graph-Exchange File Formats. *arXiv:1503.02781 [cs]*, March 2015. URL: <http://arxiv.org/abs/1503.02781>.
- [69] Kirk Saban. Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency. Technical Report WP380 (v1.2), Xilinx, December 2012. URL: https://www.xilinx.com/support/documentation/white_papers/wp380_Stacked_Silicon_Interconnect_Technology.pdf.
- [70] Abolghasem Sadeghi-Niaraki, Masood Varshosaz, Kyehyun Kim, and Jason J. Jung. Real world representation of a road network for route planning in GIS. *Expert Systems with Applications*, 38(10):11999–12008, September 2011. DOI: 10.1016/j.eswa.2010.12.123. URL: <https://linkinghub.elsevier.com/retrieve/pii/S0957417410014867>.
- [71] Sherif Sakr. *Encyclopedia of Big Data Technologies*. Springer Berlin Heidelberg, New York, NY, 2019.
- [72] SciPy. SciPy v.1.6.0 Reference Guide: Sparse matrices, December 2020. URL: <https://docs.scipy.org/doc/scipy/reference/sparse.html>.

- [73] Zhiyuan Shao, Ruoshi Li, Diqing Hu, Xiaofei Liao, and Hai Jin. Improving Performance of Graph Processing on FPGA-DRAM Platform by Two-level Vertex Caching. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 320–329, Seaside CA USA, February 2019. ACM. DOI: 10.1145/3289602.3293900. URL: <https://dl.acm.org/doi/10.1145/3289602.3293900>.
- [74] Linda L. Shen, Ibrahim Ahmed, and Vaughn Betz. Fast Voltage Transients on FPGAs: Impact and Mitigation Strategies. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 271–279, San Diego, CA, USA, April 2019. IEEE. DOI: 10.1109/FCCM.2019.00044. URL: <https://ieeexplore.ieee.org/document/8735575/>.
- [75] Shuaiwen Song, Rong Ge, Xizhou Feng, and Kirk W. Cameron. Energy Profiling and Analysis of the HPC Challenge Benchmarks. *The International Journal of High Performance Computing Applications*, 23(3):265–276, August 2009. DOI: 10.1177/1094342009106193. URL: <http://journals.sagepub.com/doi/10.1177/1094342009106193>.
- [76] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPOPP '13*, page 135, Shenzhen, China, 2013. ACM Press. DOI: 10.1145/2442516.2442530. URL: <http://dl.acm.org/citation.cfm?doid=2442516.2442530>.
- [77] Pedro Filipe Silva, João Bispo, and Nuno Paulino. Building Beyond HLS: Graph Analysis and Others. In *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE '21)*, Virtual USA, April 2021. URL: <https://capra.cs.cornell.edu/latte21/paper/15.pdf>.
- [78] Tyler Stocksdales, Mu-Tien Chang, Hongzhong Zheng, and Frank Mueller. Architecting HBM as a high bandwidth, high capacity, self-managed last-level cache. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems - PDSW-DISCS '17*, pages 31–36, Denver, Colorado, 2017. ACM Press. DOI: 10.1145/3149393.3149394. URL: <http://dl.acm.org/citation.cfm?doid=3149393.3149394>.
- [79] Leo Sun. Intel Should Worry About AMD’s Rumored Interest in Xilinx. *The Motley Fool*, October 2020. URL: <https://www.fool.com/investing/2020/10/14/intel-worry-about-amd-rumored-interest-xilinx/>.
- [80] Ananta Tiwari, Michael A. Laurenzano, Laura Carrington, and Allan Snaveley. Modeling Power and Energy Usage of HPC Kernels. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 990–998, Shanghai, China, May 2012. IEEE. DOI: 10.1109/IPDPSW.2012.121. URL: <http://ieeexplore.ieee.org/document/6270746/>.
- [81] Stephen M. Trimberger. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE*, 103(3):318–331, March 2015. DOI: 10.1109/JPROC.2015.2392104. URL: <http://ieeexplore.ieee.org/document/7086413/>.

- [82] David Vojtek, Lukáš Rapant, Martin Golasowski, Jiří Ševčík, Vít Ptošek, Kateřina Slani-
nová, and Jan Martinovič. Traffic network routing index for the Czech Republic, Decem-
ber 2018. DOI: 10.5281/ZENODO.2275556. URL: [https://zenodo.org/record/
2275556](https://zenodo.org/record/2275556).
- [83] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang.
SEP-graph: Finding shortest execution paths for graph processing under a hybrid frame-
work on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of
Parallel Programming*, pages 38–52, Washington District of Columbia, February 2019.
ACM. DOI: 10.1145/3293883.3295733. URL: [https://dl.acm.org/doi/10.1145/
3293883.3295733](https://dl.acm.org/doi/10.1145/3293883.3295733).
- [84] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and
John D. Owens. Gunrock: A high-performance graph processing library on the
GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and
Practice of Parallel Programming*, pages 1–12, Barcelona Spain, February 2016.
ACM. DOI: 10.1145/2851141.2851145. URL: [https://dl.acm.org/doi/10.1145/
2851141.2851145](https://dl.acm.org/doi/10.1145/2851141.2851145).
- [85] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. En-
ergy Efficient Scientific Computing on FPGAs using OpenCL. In *Proceedings of the 2017
ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 247–
256, Monterey California USA, February 2017. ACM. DOI: 10.1145/3020078.3021730.
URL: <https://dl.acm.org/doi/10.1145/3020078.3021730>.
- [86] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Day. Virtex UltraScale+
HBM FPGA: A Revolutionary Increase in Memory Performance. Technical Report WP485
(v1.1), Xilinx, July 2019. URL: [https://www.xilinx.com/content/dam/xilinx/
support/documentation/white_papers/wp485-hbm.pdf](https://www.xilinx.com/content/dam/xilinx/support/documentation/white_papers/wp485-hbm.pdf).
- [87] Wayne Wolf. *FPGA-Based System Design*. Prentice Hall Modern Semiconductor Design
Series. Prentice Hall, Upper Saddle River, N.J, 4. print edition, 2004.
- [88] Xilinx. Xilinx Extends Data Center Leadership with New Alveo U280 HBM2
Accelerator Card; Dell EMC First to Qualify Alveo U200, November 2018.
URL: [https://www.xilinx.com/news/media-kits/xilinx-extends-
data-center-leadership-with-new-alveo-u280-hbm2-accelerator-
card.html.html#photos](https://www.xilinx.com/news/media-kits/xilinx-extends-data-center-leadership-with-new-alveo-u280-hbm2-accelerator-card.html.html#photos).
- [89] Xilinx. Alveo U280 Data Center Accelerator Card User Guide. Technical Re-
port UG1314 (v1.2.1), Xilinx, November 2019. URL: [https://www.mouser.com/
pdfDocs/u280userguide.pdf](https://www.mouser.com/pdfDocs/u280userguide.pdf).
- [90] Xilinx. QDMA Subsystem for PCI Express v3.0. Technical Report PG302 (v3.0), Xil-
inx, September 2019. URL: [https://www.xilinx.com/support/documentation/
ip_documentation/qdma/v3_0/pg302-qdma.pdf](https://www.xilinx.com/support/documentation/ip_documentation/qdma/v3_0/pg302-qdma.pdf).
- [91] Xilinx. Alveo U280 Data Center Accelerator Card Data Sheet. Technical Report
DS963 (v1.3), Xilinx, May 2020. URL: [https://www.xilinx.com/support/
documentation/data_sheets/ds963-u280.pdf](https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf).
- [92] Xilinx. PYNQ: Python Productivity for ZYNQ, October 2020. URL: [https://
github.com/xilinx/pynq](https://github.com/xilinx/pynq).

- [93] Xilinx. Vitis Graph Library: Internal Design of Pagerank, 2020. URL: https://xilinx.github.io/Vitis_Libraries/graph/2020.2/guide_L2/kernels/PageRank.html.
- [94] Xilinx. Vitis Graph Library: Internal Design of PagerankMultiChannels, 2020. URL: https://xilinx.github.io/Vitis_Libraries/graph/2020.2/guide_L2/kernels/PageRankMultiChannels.html.
- [95] Xilinx. Vitis Graph Library: Internal Design of Single-Source Shortest Path, 2020. URL: https://xilinx.github.io/Vitis_Libraries/graph/2020.2/guide_L2/kernels/ShortestPath.html.
- [96] Xilinx. Alveo Data Center Accelerator Card Platforms User Guide. Technical Report UG1120 (v1.5), Xilinx, July 2021. URL: https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/ug1120-alveo-platforms.pdf.
- [97] Xilinx. Alveo Platform Loading Overview, September 2021. URL: https://xilinx.github.io/XRT/master/html/platforms_partitions.html.
- [98] Xilinx. AXI High Bandwidth Memory Controller v1.0 LogiCORE IP Product Guide. Technical Report PG276 (v1.0), Xilinx, August 2021. URL: https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf.
- [99] Xilinx. Getting Started with Alveo Data Center Accelerator Cards. Technical Report UG1301 (v1.9), Xilinx, July 2021. URL: https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/1_9/ug1301-getting-started-guide-alveo-accelerator-cards.pdf.
- [100] Xilinx. Vitis Application Acceleration Development Flow Documentation: Migrating to a New Target Platform, July 2021. URL: https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/migrating_new_target_platform.html.
- [101] Xilinx. Vitis High-Level Synthesis User Guide. Technical Report UG1399 (v2020.2), Xilinx, March 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf.
- [102] Xilinx. Vitis Libraries, 2021. URL: https://github.com/Xilinx/Vitis_Libraries.
- [103] Xilinx. Vitis Libraries v2021.1, June 2021. URL: https://xilinx.github.io/Vitis_Libraries/.
- [104] Xilinx. Vitis Tutorials: HBM Overview, August 2021. URL: https://xilinx.github.io/Vitis-Tutorials/master/docs/docs/Runtime_and_System_Optimization/Feature_Tutorials/04-using-hbm/1_overview.html.
- [105] Xilinx. Xilinx Runtime (XRT) Architecture: Host Memory Access, September 2021. URL: <https://xilinx.github.io/XRT/master/html/hm.html>.

- [106] Chengbo Yang, Long Zheng, Chuangyi Gui, and Hai Jin. Efficient FPGA-based graph processing with hybrid pull-push computational model. *Frontiers of Computer Science*, 14(4):144102, August 2020. DOI: 10.1007/s11704-019-9020-5. URL: <http://link.springer.com/10.1007/s11704-019-9020-5>.
- [107] Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin, Ling Liu, and Kisung Lee. Fast Iterative Graph Computation: A Path Centric Approach. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 401–412, New Orleans, LA, USA, November 2014. IEEE. DOI: 10.1109/SC.2014.38. URL: <http://ieeexplore.ieee.org/document/7013020/>.
- [108] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. GraphIt: A high-performance graph DSL. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, October 2018. DOI: 10.1145/3276491. URL: <https://dl.acm.org/doi/10.1145/3276491>.
- [109] Ruohuang Zheng and Sreepathi Pai. Efficient Execution of Graph Algorithms on CPU with SIMD Extensions. In *2021 IEEE/ACM Int. Symp. Code Gener. Optim.*, pages 262–276, Seoul, Korea (South), February 2021. IEEE. DOI: 10.1109/CGO51591.2021.9370326. URL: <https://ieeexplore.ieee.org/document/9370326/>.
- [110] Shijie Zhou, Charalampos Chelmiss, and Viktor K. Prasanna. High-Throughput and Energy-Efficient Graph Processing on FPGA. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 103–110, Washington, DC, USA, May 2016. IEEE. DOI: 10.1109/FCCM.2016.35. URL: <http://ieeexplore.ieee.org/document/7544758/>.
- [111] Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman, and Qing Wu. HitGraph: High-throughput Graph Processing Framework on FPGA. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2249–2264, October 2019. DOI: 10.1109/T-PDS.2019.2910068. URL: <https://ieeexplore.ieee.org/document/8685122/>.
- [112] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K. Prasanna. An FPGA framework for edge-centric graph processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 69–77, Ischia Italy, May 2018. ACM. DOI: 10.1145/3203217.3203233. URL: <https://dl.acm.org/doi/10.1145/3203217.3203233>.
- [113] Shijie Zhou and Viktor K. Prasanna. Accelerating Graph Analytics on CPU-FPGA Heterogeneous Platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 137–144, Campinas, October 2017. IEEE. DOI: 10.1109/SBAC-PAD.2017.25. URL: <http://ieeexplore.ieee.org/document/8102188/>.
- [114] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 409–420, Salt Lake City, UT, November 2016. IEEE. DOI: 10.1109/SC.2016.34. URL: <https://ieeexplore.ieee.org/document/7877113/>.